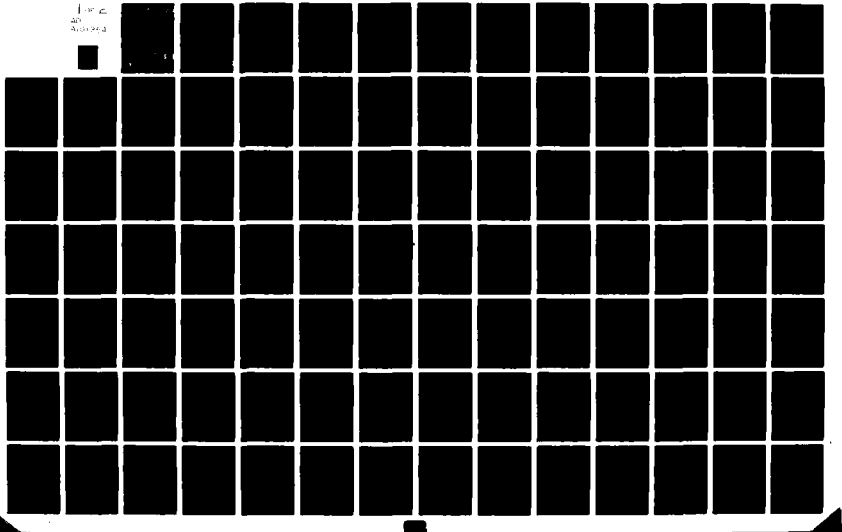
AD-A101 954 ROCHESTER UNIV NY DEPT OF COMPUTER SCIENCE F/G 9/2
PERFORMANCE EVALUATION OF COMMUNICATING PROCESSES.(U)
MAY 80 I GERTNER N00014-78-C-0164
UNCLASSIFIED TR-76 NL



LEVEL II

(12)

AD A101954

DTIC
ELECTE
JUL 24 1981
S
D

Department of Computer Science
University of Rochester
Rochester, New York 14627

DTIC FILE COPY

81 6 15 168

LEVEL II

PERFORMANCE EVALUATION of COMMUNICATING PROCESSES,

TR76
May 1980

Ilya Gertner
Computer Science Department
University of Rochester
Rochester, New York

Submitted in Partial Fulfillment
of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY

The work reported in this thesis was partially funded by DARPA Grant N00014-78-C-0164.

Curriculum Vitae

Ilya Gertner was born in Kaunas, Lithuania, U.S.S.R. on April 20, 1950. He attended Polytechnical Institute of Kaunas but received his B.S. in 1973 from Technion, Israel. After serving in the Israeli army, he continued in graduate school at Technion, receiving his M.S.

He entered the University of Rochester in the fall of 1976. At the Computer Science Department he has been involved in the development of RIG. His interests include running, bicycling, skiing and chess.

Acknowledgments

The author would like to express his appreciation to his advisor Paul Rovner and members of his Ph.D committee: Professors Jerome Feldman and James Low. Each of the committee members contributed in a significant way to the dissertation.

Appreciation also goes to Rose Peet for translating my "personal dialect" into English.

And, finally, to all students who participated in the development of RIG, without which this thesis would be impossible.

ABSTRACT

Understanding the behavior of communicating processes is essential to the evaluation of distributed operating systems. This dissertation focuses on performance analysis of existing distributed systems using finite state machine models of computation. The performance evaluator describes a finite state machine that represents a particular abstraction, the system of interest. Different finite state machines may be formulated and applied to the same measurement data to extract different kinds of information. To test the ideas in the environment of our local network, I have implemented a performance-monitoring system that was used to analyze RIG, a message-based distributed operating system. This required a language for describing finite state machines using symbolic references to RIG processes, messages and a hierarchy of finite state machines. Elementary finite state machines describe the behavior of a single process representing a sequential program. Composite finite state machines describe a group of communicating processes representing a parallel program. The behavior of a sequential program is characterized by a total ordering of events; the behavior of a parallel program is characterized by a partial ordering. Representing all the possible orderings of events in the composite model is an intractable task. In our experience with RIG, such a composite model includes a great many paths which almost never occur. The challenge, therefore, is to find those paths that occur often in the execution of the system and are of significant duration. To aid the performance evaluator in describing these paths, I introduce three new kinds of transitions: the first characterizes a long sequence of messages; the second describes the overall system state as a vector of process states; the third describes a limited number of messages in a stream. This is a novel idea in describing composite models of computation. Although we analyze examples only from the RIG system, many ideas can be applied to other programs that are characterized by sequential behavior at some level of abstraction.

CONTENTS

List of Figures

1. Overview and Outline

1.1 Introduction

Measuring the performance of communicating processes is difficult due to the partial ordering of events. I introduce a total ordering of events in the context of a finite state machine model. Different finite state machines may be formulated and applied to the same data to extract different kinds of information. The basic assumption used throughout the dissertation is that even parallel programs are characterized by sequential behavior at some level of abstraction. The major concern is the development of programming tools and a methodology for performance monitoring of distributed systems.

This dissertation was primarily motivated by the author's experience in tuning the RIG system (Rochester's Intelligent Gateway for the local network at the Computer Science Department) [Ball, Burke, Gertner, Lantz, and Rashid, 1979]. RIG is a message-based distributed operating system intended to serve as an intermediary between the human user and a variety of computing facilities on the local network. The work reported herein is a child of this environment: The performance monitoring system ("the monitor") runs on a stand-alone minicomputer used to collect statistics of the distributed operating system. Messages are the basic events being measured (if a higher resolution is required, a process must be modified to send a pseudo-message). Finite state machines are used by the monitor to select events of interest and to present results to a performance evaluator. Events of interest are those messages that trigger state-transitions in a finite state machine model. The performance evaluator is a person using programming tools in an attempt to understand the performance of a running system. The quality of RIG has improved substantially due to the implementation and use of the monitor. More to the point, the suitability of the mechanisms and underlying principles of the performance monitoring system was tested by real measurements on a real system.

There are three general purposes of performance evaluation: selection evaluation, performance projection, and performance monitoring [Lucas 71]. Selection evaluation uses performance as the major criterion in the decision to obtain a particular system from a vendor. Performance projection is oriented toward designing a new system. The goal in performance projection is to estimate the performance of a system that does not yet exist. Performance monitoring provides data on the actual performance of an existing system. It is generally used to locate a bottleneck limiting performance when either reconfiguring the existing hardware or improving the execution of software. There are two types of performance

monitoring: sampling and event tracing. Sampling monitors initiate data collection activities when a real time clock signals the end of an interval. The interval or sampling period is usually constant. The time overhead of sampling monitors is minimal and fixed.

Event tracing monitors usually obtain more detailed information about system operation over a shorter period of time. At the occurrence of a prescribed event, the control of the computer operating system is passed to the event tracing monitor. The events are collected and recorded for subsequent analysis. Two major problems of event tracing monitors are: 1) accumulation of vast data in a short period of time and 2) significant overhead in the system caused by the data collection. Both problems were avoided in the RIG system. Hence, event tracing proved to be valuable and practical for measuring performance. Event tracing was useful both for system debugging and performance analysis. In the case of a software error, the event trace helped to understand conditions under which the error occurred in the system. (A similar experience has also been reported with other operating systems [Lausen 75]).

There is a wide range in the possible levels of program abstraction for the purpose of producing event traces. One extreme is measuring execution of every instruction; the opposite extreme is measuring the entire program as a single operation. Neither of two extremes is a useful characterization of the program's behavior. To find the right level of abstraction is a very difficult problem; in the case of multi-process systems in which processes communicate via messages, the natural choice is measuring messages. In RIG, messages are higher-level constructs than procedures (handling a message requires several procedure calls and possibly sending or receiving additional messages) but are more detailed than user-level activities like entering a line to the text editor (entering a line to the text editor requires four processes to exchange five messages).

I define the execution of a process as a sequence of events where an event is reception of a message or a change in a state explicitly declared by the process. Every event carries several time stamps that are used to compute various intervals of interest to the user. Analysis of the message trace is difficult due to the large amount of data and arbitrary ordering of messages. Although statistical data reduction techniques help to gather a profile on the use of the system [Lucas 71], or to estimate parameters for queueing network models [Rose 78], analysis of RIG has not benefitted from the statistics of a particular message. The main difficulty is a lack of the context or conceptual framework within which to evaluate those statistics. Other systems provide general purpose data reduction packages suiting various users ([MacDougall 78] and [McDaniel 77]).

In this dissertation I introduce a conceptual framework based upon a finite state machine model of computation.

In RIG, the performance evaluator analyzes a message trace in two steps. First, he measures the system at macro level; for example, it takes about 2 seconds for the system to respond to a user entering a new line in a file. Second, he searches for a set of measurements that give him an objective view consistent with the previous evaluation of the system. One possible explanation is the following: 1 second was spent in handling requests of other users, 1/2 second in handling the user's terminal keyboard and screen, and 1/2 second in communicating with the file system. The performance evaluator might choose to obtain more detailed measurements on the way in which characters are displayed on the screen. A formal definition of an abstract model describing those events of interest and an automatic system to analyze raw statistics in light of the model would be great assets to the performance evaluator.

System measurements in terms of abstract models have been used before for the purpose of estimating parameters of simulation models like E-nets [Nutt 72], or Petri nets [Peterson 77] but not for performance monitoring. (Simulation models are beyond the scope of this dissertation; the last chapter discusses some possible future work in this direction.) Those models are well suited to describing highly parallel structures but are difficult to understand for the performance evaluator who is dealing with the real system. The main difficulty is understanding the statistics of parallel transitions that are only partially ordered in time.

Related research in the areas of design and verification of correctness of communicating processes has developed a formalism for the analysis of message traces. The formalism is based upon finite state machines ([Bochman 78], [Feldman 77], and [West 78]). Similar formalism can then be applied to performance analysis of communicating processes. The main advantage in using finite state machines is simplicity due to the total ordering of events in the context of the model; the major problem- the large number of states in the composite models- is dealt with in this dissertation.

Describing accurate models of computation is an art. Many experiments, as well as deep understanding of the system, are required to debug the model of a computation. To support those experiments in the context of RIG, I have implemented and used a language that describes various finite state machines for RIG. This language uses symbolic references to RIG processes, messages, and to a hierarchy of finite state machines. Elementary finite state machines describe the behavior of a single process representing a sequential program. Composite finite state machines describe a group of communicating processes representing a

parallel program.

The main difficulty in describing composite models for communicating processes is to recognize all system states (a vector of process states) and all the possible state-transitions. The system designer who verifies the correctness of communicating processes must examine all the possible system states even those that occur with a very low probability [Sunshine 78]; the performance evaluator who analyzes the performance of communicating processes examines only a small subset of states that occur often in the execution of the system and are of significant duration. These states represent the user view of computations in the system. In RIG, the number of states which are actually reached during execution of the system is very small. Under changing load conditions, the system passes through a large number of states and quickly stabilizes to a new set of states characterizing the system under each new load. The surprising result was that the total number of states describing the average behavior of the system remained small for a wide range of the load.

This dissertation supports the position that finite state machines are valuable models for designing, implementing and performance analysis of communicating processes. Although I use examples only from the RIG system, many ideas can be applied to other operating systems that have well-defined interprocess communication facilities. These systems are implemented in a style which is very close in spirit to either a message-based model or to a procedure-based model [Lauer et al., 79].

The procedure-based model is characterized by a large number of very small processes, rapid creation and deletion of processes, communication by means of direct sharing and interlocking of data in memory, and identification of the context of execution with the function rather than with the process. In a message-based system, synchronization among processes and queueing for congested resources is implemented in message queues attached to the process associated with those resources. In a procedure-based system, synchronization occurs in a form of queues of processes waiting for locks associated with the corresponding data structures. In this dissertation, each message is marked with three time stamps: (1) the time the sender has queued the message; (2) the time the receiver has accepted the message; (3) the time the receiver completed processing of the message. These time stamps are used to compute various intervals of interest to the performance evaluator. Likewise, three time stamps are sufficient for a procedure-based system: (1) the time a process has accessed a lock guarding a shared resource; (2) the time the process has obtained control of the lock; (3) the time the process releases the lock. Analogous intervals can then be compted for each procedure call.

Modeling and analysis of complex systems which exhibit concurrent behavior requires various automated (computer aided) tools. Computer aided design which is based on the development of machine processable models and the use of computer tools to evaluate those models have shown promise [Estrin et al., 79]. This dissertation applies similar models for the performance analysis of existing distributed systems. Future systems will be designed, implemented and documented using formal models of computation. The same models (or simplified) can then be applied for performance analysis of these systems.

1.2 Related Research

This dissertation relates to three areas: (1) performance monitoring of operating systems since the examples of communicating processes are taken from the real operating system; (2) design and verification of correctness of communicating processes because finite state machines have been used to verify the correctness of communication protocols; (3) message-based computing because the potential application of the methods developed in this dissertation depends upon the future use of the message-passing discipline.

Performance monitoring of running programs proceeds in two phases [Lucas 71]: First, an execution trace that contains events of interest is generated; second, various statistics are calculated from the trace to provide the user with an insight about the program's behavior. This methodology was applied to the analysis of page references in ALGOL programs [Batson 76]. The novel feature was the high-level of a program's instrumentation for the purpose of statistics gathering. We should not expect the programmer to debug and optimize the performance of his program through the use of memory dumps, loader maps, machine addresses and similar diagnostic tools. Rather, our new systems should be engineered as complete high-level language machines in which all diagnostic information is presented in terms of the symbolic source language as written by the programmer. These principles are used in this dissertation by developing a high-level language interface for the performance evaluator who uses a similar language and the same symbols both for programming and performance analysis.

Earlier work on the analysis of trace data also used a graph model of the system. Some graphs were defined using the source code of a program at the level of a machine instruction [Howard and Alexander, 72]. To reduce the complexity of graphs, the authors introduced a hierarchy of graphs and considered only a limited set of instructions (check points within the program). The correct selection of check points (which is very difficult and is not automated) was vital to the successful construction of those graph models. Other graphs were produced using a trace of the program execution produced by a probe in the operating system itself [Anderson 76]. The system recorded a trace of events at the job level, e.g. starting an input operation, running the job, or waiting for the completion of a swap operation. Here, automatic construction of graph models worked better due to the linear structure of graphs; user intervention was still required to eliminate paths in the graph model that occurred very seldom. These paths did not contribute to better models but significantly complicated it. In this dissertation, I use graph models of computations but make no attempt to automate construction of the graphs (although I develop a high-level language to

describe those graphs).

Many systems support very general data reduction packages. Most computer manufacturers provide a General Program Trace Facility. The trace facility combined with a high-level language describing events of interest was found extremely useful for the analysis of existing systems [MacDougall 78]. Yet another degree of flexibility in collection and analysis of measurements was achieved for a personal computer system connected to a local network [McDaniel 77]. Data collection and analysis is performed on different machines at different times thereby reducing the impact of measurements on the running programs. I use a similar architecture with the performance monitoring system running on a separate computer to collect statistics of other computers connected to a local network.

In summary, data collection and analysis is still an art: There are no rules for the choice of the "right" level of abstraction for the purpose of measurements; similarly, there are no rules for what to do with the data. As a result, some systems support very general data collection and reduction programs that postpone the burden of decisions to the system's users (for example, [MacDougall 78] and [McDaniel 77]). Fortunately, in the case of multi-process systems in which processes communicate via messages, we have much better intuitions on how to characterize the behavior. Dealing with message traces is a natural choice for such systems. Data collection is easy because there is a small number of central system routines that support interprocess communication. Data analysis is better understood because there is an experience in using message traces for design and verification of correctness of communicating processes ([Estrin et al., 78], [Bochman 78], [Feldman 77] and [West 78]).

There is a long history of the use of state-transition models for the analysis of concurrent processes. Dijkstra introduced state variables to synchronize sequential communicating processes [Dijkstra 1966]. A state variable is an additional programming concept (a new type variable) which is used solely for the purpose of synchronization. In addition, Dijkstra advocated the design of programs to be guided by the use of these state variables. He said,

> "In my experience, one starts with a rough picture of both programs and state variables, then he starts to enumerate different states and finally tries to build the programs".

Feldman used the state notion to develop a mathematical formalism for verifying the correctness of communicating processes [Feldman 77]. He noted,

> "We can only verify (and understand) systems that have some stable state transitions of at least a subset of the modules".

To describe a model for a group of processes, the performance evaluator searches for a small number of system states (defined to be a vector of process states) that occur often in execution of the system and are of significant duration. This is achieved by either enumerating all state variables as suggested by Dijkstra, or searching for the system stable state-transitions as described by Feldman.

The related efforts listed above considered only the design specifications of communicating processes; no attempt has been made to analyze the performance of existing systems using a finite state machine model of computation. The related efforts listed below deal with state-transitions semantics that are used for the analysis of message-based computing. A notable example is SARA, a simulation system used for the design and analysis of multiprocess systems [Estrin et al., 78]. In SARA the analysis of control structures is performed with UCLA graphs (which are equivalent to Petri nets and are used to verify the correctness of control structures of the system). A system that is designed with SARA can be described with a finite state machine obtained from the UCLA graphs.

Message-based models have been used for simulation of existing systems [Chany et al., 79]. The authors analyzed a multi-process system in which processes communicate via shared memory and replaced interprocess communication instances with messages. The result was a very accurate simulation model. In addition, they developed a mathematical formalism for describing a process as a function of its variables and incoming messages. In this dissertation, a process is a finite state machine where states abstract the content of local variables.

The growing interest in message-passing suggests that many future systems will be implemented or at least designed using this discipline. Further, the developed techniques of using finite state models for the performance analysis of operating systems will be applied to a wider range of systems.

1.3 Outline of the Dissertation

Chapter Two describes the environment of communicating processes in RIG where the experiments are conducted (the experiments are described in Chapter Four). RIG can be thought of as a model of distributed computation, processes communicate only by messages and there is no shared data. The implementation and use of messages in RIG are described in detail to help the reader understand the experiments.

Chapter Three describes a formalism for the performance analysis of communicating processes. First, I define the basic properties of a message trace and the time intervals associated with each message. Next, I introduce a finite state machine model of computation. The time intervals that are used to characterize a message are then extended to characterize an event that is defined by a state-transition in the finite state machine model. The finite state machines are encoded in a language that uses symbolic references to the RIG processes, messages and a hierarchy of finite state machines.

Elementary finite state machines describe the behavior of a single process; composite finite state machines describe a group of processes. To reduce the number of states in the composite model, I use new kinds of transitions allowing one to describe a small subset of system states. The chapter uses a simplified example of a distributed graphics application to illustrate the formalism.

Chapter Four contains examples of finite state machines modeling computations in RIG. Different finite state machines are formulated and applied to the same measurement data to extract different kinds of information. One finite state machine express better the overlap between execution of parallel processes; another finite state machine express better the system overhead. On the basis of the information, the performance evaluator then points out various bottlenecks in the system. The chapter presents results which indicate the value of finite state machines for the performance analysis of communicating processes. Informally, on the basis of examples, I suggest a methodology for describing finite state machine models of computations. The presentation is based on two examples: large scale computations of many processes communicating in full hand-shake and pipelined computations of a few processes streaming messages in one direction.

Chapter Five describes the implementation of a performance monitoring system for RIG.

Chapter Six demonstrates how finite state machines are applied to entirely different areas: validation of reliable transmission protocols and optimized implementation of

high-level protocols. These two examples further support the position that finite state machines are valuable models for designing, implementing and performance analysis of communicating processes.

Chapter Seven concludes the thesis. It summarizes the experience of applying a finite state machine model to the problem of evaluating the performance of systems composed of communicating processes. Both practical results and general principles are reviewed. Contributions to the general area of understanding of communicating processes are also discussed.

The appendix contains a description of the finite state machines and a display of the time intervals in the form that is actually used by the performance monitoring system for the analysis of RIG. The example described in Section 4.1 (the initialization of a terminal in RIG) is presented in detail.

2. The Environment of RIG

2.1. Introduction

This chapter describes the RIG system for which the performance monitoring system is implemented and used to test the ideas described in this dissertation. To understand better the kind of computations considered, we describe the implementation and use of the message style of communication in RIG. By comparing RIG with other operating systems, we suggest that message-passing is a useful strategy both for the design and implementation of operating systems. The use of finite state machines for the performance analysis of this kind of computation is described in the next chapter.

This chapter has four major sections: Section 2.2 gives an overview of the RIG system and its hardware configuration. Section 2.3 describes the implementation of messages. Interrupt messages, flow control and network communications are described in detail. Section 2.4 describes the use of the message style in RIG. Finally, Section 2.5 compares RIG with other systems with emphasis on the fundamental properties of the message-passing discipline.

2.2 Overview

The first version of the RIG system was up and running in early 1976 [Ball et al., 76]. RIG was built to serve as an intermediary between the human user (working through a display terminal or personal computer) and a variety of large computer systems. The bulk of the user's computational requirements, such as user program execution and special services, is met by these large systems, which are partially integrated into the RIG system through a fast local network. RIG provides a user with basic services such as printing, plotting, local file storage, text-editing, and virtual terminal facility [Lantz and Rashid, 1979].

The following computing facilities are connected to the local network: four personal computers (Xerox Altos), two service machines (Data General Eclipses), and two time-sharing systems (DEC-10 and VAX). The minicomputers and the VAX are connected via a 3 MHz broadcast network (EtherNet). The DEC-10 communicates over a 50 KHz synchronous line to one of the two

---

This chapter is based on the paper "Perspective on Message-based Distributed Computing" by myself and other members of the RIG group [Ball et al., 79] and on the internal document "RIG System Kernel" [Gertner 79c].

Eclipses. The RIG system runs on the Eclipse computers [Figure 1].

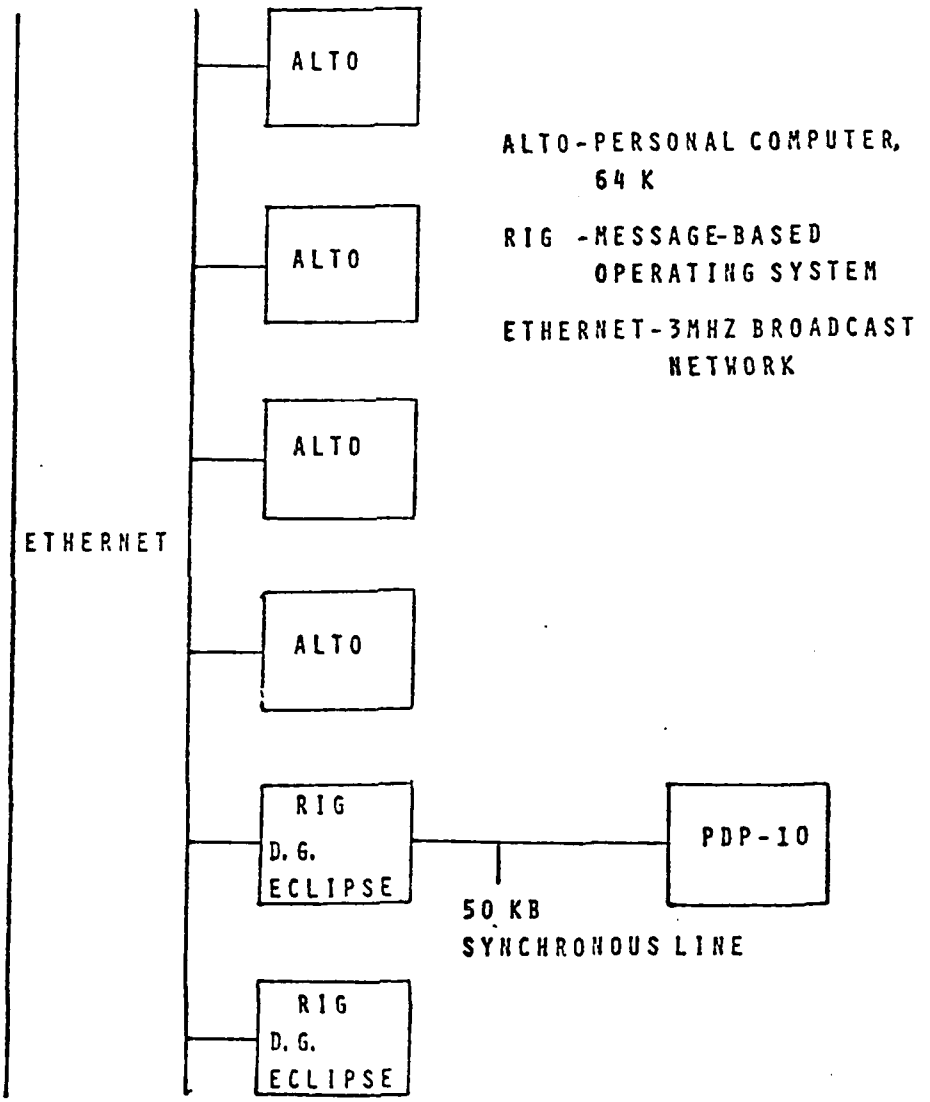


Figure 1: Environment of RIG

Logically, RIG can be thought of as a collection of independent processes running on various computers and cooperating via messages. Each RIG machine has its own kernel which provides the support functions of message-passing, process scheduling, physical memory management, and interrupt handling. Each RIG process performs a specific set of functions and has a distinct logical address.

Communication between processes takes the form of messages queued separately by the system kernel for each destination. A destination in RIG is specified by a process-port pair, where a port is simply a sub-address within a process. Ports are used for selective message reception, multiplexing and flow control. (Section 2.3 discusses the flow control mechanisms employed in RIG.)

Each system resource, such as the file system, is managed by one or more server processes which are responsible for performing resource-specific functions and for providing a standard message interface to other RIG processes.

Three aspects of the communication techniques used in RIG eliminate the need to know the actual location of services in the distributed system:

1. all basic services are provided by RIG processes through the use of messages (no shared memory);

2. remote processes send and receive messages in the same way as do local processes;

3. inter-process communication can be initiated symbolically.

The key component of the RIG design was the decision to provide a uniform interface to all system services through the use of messages. The RIG kernel serves only to provide the abstractions of process, message, and message queues. Other functions, such as file access, terminal communication, and printing, are provided by RIG processes and are made available through messages.

Thus, the distinction made in typical systems between operating system services and user processes has been abandoned. Although interprocess communication was well understood when the initial design for RIG was formulated (and had been implemented in a number of major operating systems -- Elf, Hydra, TOPS-10, Tenex, B6700 MCP), such a total dependence on message-passing was a considerable deviation from the norm.

Resource independence is achieved through the use of standardized server protocols (see Section 2.4). These

provide a consistent mechanism for opening, closing, reading, and writing entities such as files, virtual terminals, and line printers. The advantage of message-passing over abstractions provided by other operating systems for communication and device independence (e.g. Unix pipes) lies in the wider range of synchronization strategies available and the flexibility of messages to convey control and data information and to signal exceptions.

The ancestors of RIG are the inter-process communication facilities of the SAIL programming language (which had been successfully used in the Stanford Hand-Eye Project [Feldman and Sproull 1971]) and the work of Walden [Walden 1972].

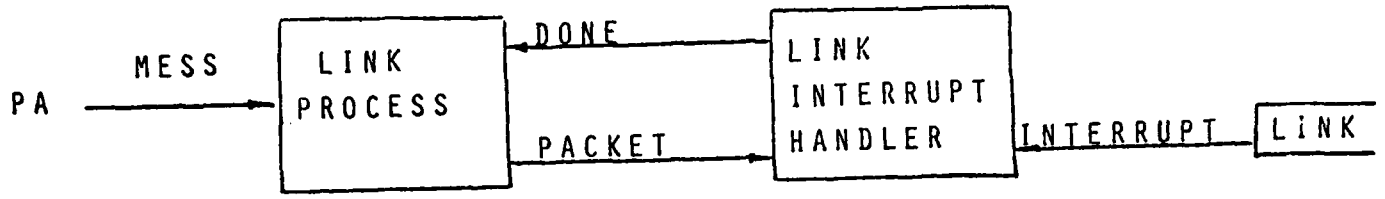
Several systems provide facilities similar to some of those provided by RIG. DEMOS [Baskett, Howard, and Montague 1977], Roscoe [Solomon and Finkel 1978], and Thoth [Cheriton, et al. 1979] are examples of systems built entirely on the use of processes communicating via messages. Other distributed systems like DCN [Mills 1976], and MSG [NSW 1976] perform computations similar to RIG.

2.3 Implementation of Messages

2.3.1 Interrupt Messages

The RIG system handles each device with two programs: the device handling process and device interrupt handler. Both programs communicate via messages. (For efficiency considerations, the actual implementation uses shared memory to support communications between device handling processes and interrupt handlers). The user communicates via messages with the device handling process.

Consider an example of a network link handled by the LinkInterrupt handler and Link process [Figure 2]. A process PA sends a message to the Link process which forwards it to the LinkInterrupt handler. If the device is idle, the LinkInterrupt handler immediately starts the transmission; otherwise, the "packet" is queued. Upon completion of the operation, a hardware interrupt arrives to the LinkInterrupt handler which forwards the message "done" to the Link process. The message "done" is queued with priority. In addition, the LinkInterrupt handler receives the next waiting "packet" and starts the transmission of a new message. If there are no "packets" waiting then the device becomes "idle".



LINK PROCESS — DRIVEN BY MESSAGES

LINK INTERRUPT HANDLER — DRIVEN BY INTERRUPTS

Figure 2: Handling of Devices

2.3.2 Flow Control

Every destination in RIG (process-port pair) uses primary and secondary queues to hold messages in transit. If the destination is local, the local system kernel does the queue management and flow control. If the destination is remote, the appropriate network server does the queue management and flow control (see Section 2.3.3 on *Networking*). A *primary queue* has a maximum message capacity (definable by the process). If a message is placed in a primary queue it is considered 'posted' and the sender is allowed to continue. If the primary queue is full, the message is queued in the secondary queue. Effectively secondary queues have infinite capacity.

A process can choose one of two options when sending a message. In the case of a "dedicated send", the sending process is kept suspended by its kernel until space on the primary input queue of the destination becomes available. A "send don't wait" is used in situations where this simple backpressure mechanism is unacceptable. For example, processes providing critical services cannot allow themselves to be suspended waiting for another process to receive a message. In such cases the sending process can request that the system kernel return a notice that the message cannot be sent and, further, that the system notify it when another message can be sent.

2.3.3 Network Communication

Network communication in RIG is provided by processes called network servers. Each RIG machine has at least one network server which handles the flow of messages to and from other machines.

A message sent from a local process PA to a process PB on a remote host is diverted by its kernel to the appropriate network server process [Figure 3]. The local server is responsible for routing and reliable transmission to the corresponding network server on the remote host. The remote network server, upon receipt of a message from PA, forwards the message to its final destination, PB. PA and PB remain unaware that the message was routed through the network servers. To facilitate the routing of messages to its final destination, a process number contains three fields: a host number, a system's incarnation number, and a local identifier [Feldman et al., 78].

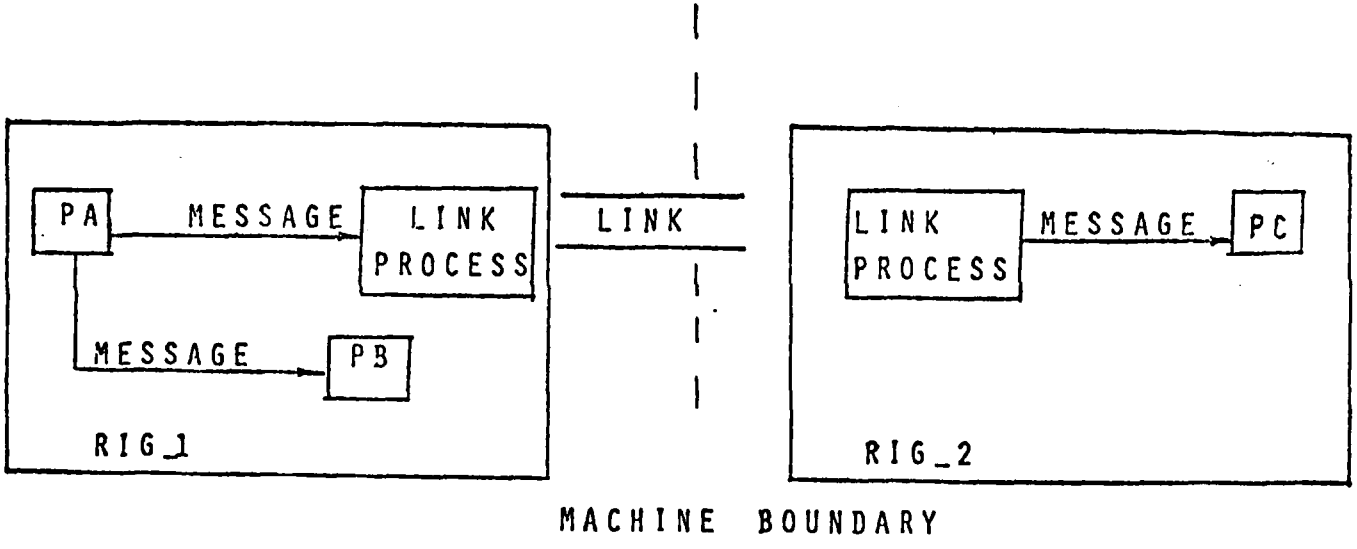


Figure 3: Network Interprocess communications

2.4 Communication Styles

When two processes wish to communicate they are free to do so in any mutually convenient manner within the constraints of the RIG message-passing primitives. In practice, we developed a set of guidelines (unenforced) that made the implementation and debugging of the system easier. We have found three fundamental styles of message communication to be sufficient:

1. atomic transactions
2. asynchronous messages
3. connections

For atomic transactions the link between the communicating processes is set up and expires on a message-to-message basis. Process PA simply composes and sends a message to process PB, without PB having to know anything about PA. Depending on the particular request, PA may or may not wait for an acknowledgment from PB. PB retains no information about PA between transactions. An example of an atomic transaction is a request for the time of day.

Certain 'interrupt' conditions (e.g. process death) are best handled as asynchronous messages not subject to normal flow control. In RIG, emergency messages provide the means for one process to alert another to the occurrence of an exceptional or unusual event. Emergency messages are queued separately and delivered when the recipient next attempts to send or receive any message. Delivery is independent of any message flow to the receiving process. Once delivered, the emergency handler (a special procedure within a process) is invoked, and is responsible for processing the event.

Any prolonged interaction between two processes (e.g. reading a file) may make it necessary for each process to remember the current state of the interaction (e.g. the file position). In such cases, the processes can create a connection by each reserving a port for subsequent interaction. Four standard procedures are conventionally used for manipulating connections -- Open, Close, Read, and Write.

Connections can be one of two types: 1) full hand-shake, or 2) streamed. Full hand-shake is, in effect, a remote procedure call [White 76]. For example, when editing a file it is necessary for the editor and file system to remain in lock-step; every transaction involving the file must be acknowledged. Full hand-shake has the advantages that the cooperating processes are always synchronized and that the initiator of the connection has complete control of the data flow. The disadvantage is the decrease in performance of the system.

Once a streamed connection has been established, the originator of data is free to transmit to the receiver without waiting for either an output acknowledgment or an input request. If the sending process can produce data faster than it can be consumed by the receiver, system defined flow control mechanisms will automatically slow down the sender (see Section 2.3.2). A typical example of streaming in RIG is copying files from one machine to another.

Streaming can be used in any situation in which a connection is established and a synchronous response to input and output requests is not necessary. The advantages of streaming are its low message overhead and the fact that it allows pipelining. The major disadvantage is that exceptional conditions must be signalled asynchronously to the flow of data, making harder to write programs and debug.

2.5 Summary

RIG is a multi-process system in which processes communicate via messages. Many other systems support a subset of similar interprocess communication facilities. In fact, the implementation of some of these systems is similar to RIG. For example, Thoth [Cheriton et al., 79] also uses a fixed message header for interprocess communication. (In contrast to RIG, however, Thoth uses shared buffer pools to communicate large amounts of data; RIG copies buffers from one process to another). Related systems are characterized by similar approach to design problems. For example, in the NSW system, the Tool Initialization Scenario is similar to the initialization of the virtual terminal in RIG (see Chapter 4).

Communication styles in RIG are characterized by full hand-shake and message-streaming. This dissertation uses finite state machines to describe this kind of computation; it is less clear, however, that the finite state machine formalism applies equally well to other interprocess communication styles (e.g shared memory models).

The advantages of systems with well-defined interprocess communication are well recognized. Several systems (SARA [Estrin et al., 1979] and DREAM [Riddle et al., 78]) have been developed to use a message-based operating system as a model for the design of any system. Although the actual implementation of a system may be based on a shared memory model, the design is characterized by the message-passing discipline. In all those cases, performance analysis of communicating processes will become of paramount interest.

3. Performance Measurements of Message Traces

3.1 Introduction

This chapter describes a formalism for the performance analysis of communicating processes. First, I define the basic properties of a message trace and the time intervals associated with each message. Next, I introduce a finite state machine model of computation. The time intervals that are used to characterize a message are then extended to characterize an event that is defined by a state-transition in the finite state machine model. The finite state machines are encoded in a language that uses symbolic references to the RIG processes, messages and a hierarchy of finite state machines.

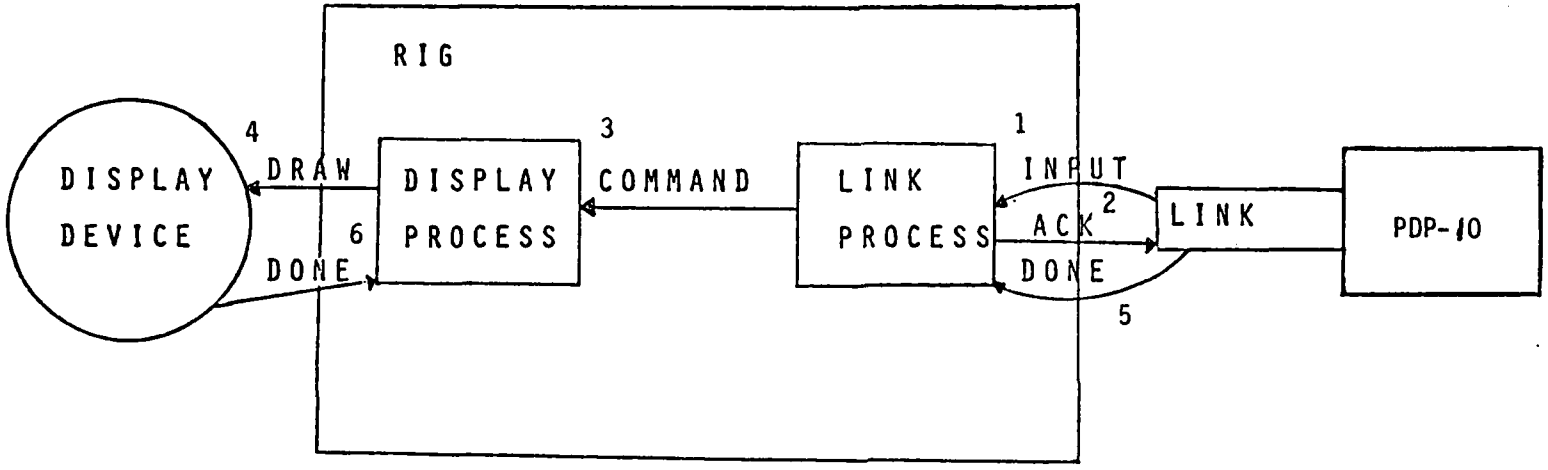
The chapter has three major sections: Section 3.2 describes in detail the example of the distributed graphics application used throughout the chapter. Section 3.3 describes the basic properties of message traces and introduces a formalism to calculate various time intervals of interest to the performance evaluator. Section 3.4 describes the language for defining finite state machine models of computation; Appendix B contains the BNF definition of a model.

This chapter uses a simplified example to illustrate the formalism; the next chapter uses real examples from the RIG system to present results supporting the position that finite state machines are practical and valuable models for the performance analysis of communicating processes. Chapter Six applies the same finite state machine formalism to two entirely different areas: validation of the behavior of reliable communications protocols and efficient implementation of higher-level protocols. The suitability of the new constructs that are developed for performance analysis of communicating processes is thus tested by applying the new constructs to different areas.

3.2 An Example of a Distributed Graphics Application

Consider an example of a distributed graphics application [Figure 4]. The PDP-10 (Digital Equip. Corp., TOPS-10 system) produces a binary representation of a picture and sends it over the link. The RIG system receives the data and displays it on the graphics device.

Portions of this chapter are described in the paper "Performance Evaluation of Communicating Processes" [Gertner 79].



LINK PROCESS- RELIABLE TRANSMISSION AND FLOW CONTROL

DISPLAY PROCESS- INTERPRETS THE COMMAND AND DRAWS ON THE SCREEN

Figure 4: Example of Distributed Graphics

Two processes and two devices in RIG are involved in this computation. The link-handling process provides reliable transmission and flow control between local and remote processes (messages 1, 2, 3, 5). The display-handling process validates the command and executes it by drawing on the graphics device (messages 4, 6). Each message in the trace contains three fields: the sender, receiver, and message identifier. The sender defines the source of the message; the receiver defines the destination; the message identifier defines the function to be executed by the receiver upon acceptance of the message.

Two user requests may produce 12 messages on the server machine running RIG. The messages appear in the order of their acceptance in this particular example. (Messages 1-6 occur for the first command; messages 7-12 for the second).

```
1)  (LinkIntInput   -> LinkProcess,    Input)
2)  (LinkProcess    -> LinkIntOutput,  Ack)
3)  (LinkProcess    -> DisplayProcess, Command)
4)  (DisplayProcess -> DisplayInt,     Draw)
5)  (LinkIntOutput  -> LinkProcess,    Done)
6)  (DisplayInt     -> DisplayProcess, Done)

7)  (LinkIntInput   -> LinkProcess,    Input)
8)  (LinkProcess    -> LinkIntOutput,  Ack)
9)  (LinkProcess    -> DisplayProcess, Command)
10) (DisplayProcess -> DisplayInt,     Draw)
11) (LinkIntOutput  -> LinkProcess,    Done)
12) (DisplayInt     -> DisplayProcess, Done)
```

Figure 5: Message Trace

The semantics of messages are described below:

1) (LinkIntInput -> LinkProcess, Input)

The hardware interrupt of the link input device signals the arrival of an input packet. The interrupt handler (the process is LinkIntInput) sends the interrupt message (the message identifier is Input) to the link handling process (the process is LinkProcess). Having received the message, LinkProcess acknowledges the foreign link handling process (message 2), decodes the message into RIG format and routes it to the local destination- the display handling process (message 3).

2) (LinkProcess -> LinkIntOutput,Ack)

The link handling process sends back an acknowledgment (Ack)

to the link output interrupt handler that started the output operation.

3) (LinkProcess -> DisplayProcess, Command)

A message of type Command arrives at the display handling process (the process is DisplayProcess) which checks for the rights of the sender and the validity of the operation, and then sends the request to the display interrupt handler (message 4).

4) (DisplayProcess -> DisplayInt,Draw)

The display interrupt handler (the process is DisplayInt) receives the message Draw and immediately starts the operation on the graphics device.

5) (LinkIntOutput -> LinkProcess, Done)

The hardware interrupt of the link output device signals completion of the link output operation (message 2). LinkIntOutput sends the message Done to LinkProcess which receives the message and releases resources associated with it.

6) (DisplayInt -> DisplayProcess,Done)

The hardware interrupt of the graphics output device signals completion of the graphics output operation (message 4). DisplayInt sends message Done to DisplayProcess which receives the message and releases buffers.

This example of distributed graphics is used throughout the chapter to illustrate the formalism. The trace of messages is used to define various time intervals of interest to the performance evaluator. Later, those messages are used to define state-transitions in finite state machines modeling different subsystems of distributed graphics.

3.3 Message Traces

3.3.1 Introduction

This section describes the basic properties of message traces and a formalism for calculating various time intervals of interest to the performance evaluator. First, I describe the time intervals associated with every message. To provide a conceptual framework within which to evaluate these intervals, I introduce the notion of a user activity- a sequence of messages implementing a given task. Analogous intervals are then defined for an activity. The time intervals of a single message are then expressed in terms of the activity. To characterize performance of parallel

processes (that run on different processors), I define new kinds of time intervals.

3.3.2 Time Intervals of a Message

This section describes the time intervals associated with every message. The system kernel produces several time stamps and statistics related to the use of system resources [Figure 7]. Independent of the operation of the system, the performance monitor calculates time intervals that are of interest to the performance evaluator [Figure 7].

Every message carries three time stamps: its birth, the time the sender has queued the message; the beginning of execution, the time receiver accepted the message; and the end of the execution, the time receiver completed processing of the message. In addition, every message carries some measure of the system overhead and the number of swapped pages. These statistics are used to compute various intervals of interest to the performance evaluator. To conveniently describe those intervals, I introduce some notation.

> A Precursor Pr(M) of message M is an occurrence of the same type of message (which is defined by the same triple) immediately before the occurrence of M.

The operations of hardware devices are marked with the same time stamps that are used to mark messages flowing between processes [Figure 7]. For messages flowing from the interrupt level, the birth of the message marks the occurrence of the hardware interrupt. For messages flowing from a process to the interrupt level of the system, the start time marks the beginning of the device operation (which is controlled by the interrupt handler) and the finish time marks the completion of the device operation. Although those time stamps are harder to obtain for hardware devices, there are many advantages in the uniformity of notation.

The delay time is the difference between the time when the receiver actually accepts the message and the time when the sender queues the message [Figure 6]. A higher degree of multiprogramming causes larger delays. The time interval between the time when the sender queues a messages and the time when the sender has queued another message of that same type (the precursor message) characterizes the frequency of incoming messages. (An alternative is to measure the interval between completions of the same type of message. For the purpose of tuning a system for stand-alone applications, I have found it to be sufficient to measure only the ratio of incoming messages). The execution time is

the difference between the time when the receiver completes processing of a message and the time when the receiver has accepted the message.

Birth(M) - time when the message was queued by the sender.

Start(M) - time when the message was accepted by the receiver.

Finish(M) - time when the receiver completed processing the message.

Overhead(M) - time that the system spends in scheduling the receiver accepting message M.

Swapped(M) - the number of pages that the system reads into main memory.

Figure 6: Event Time Stamps

Delay(M) = Start(M) - Birth(M)

The time that the message spends in in the input queue of the receiver.

Interval(M) = Birth(M) - Birth(Pr(M))

The time interval between the current message and the last occurrence of the same type of message.

Execution(M)= Finish(M) - Start(M)

The time required by the receiver to handle that message.

Figure 7: Time intervals of a message

3.3.3 Statistics of Messages

To obtain statistics of messages, the performance evaluator describes triples of messages using symbolic references to RIG processes and messages. The same symbols are used both for programming of communicating processes and for describing triples of messages that are measured. The following triple

(LinkIntInput -> LinkProcess, Input)

accumulates statistics for each packet arriving from LinkIntInput to LinkProcess. The statistics include accumulated time values and histograms (other statistical parameters can also be computed). To collect statistics for a broader class of messages matching the specified pattern, I introduce a new construct, ANY. For example, the following triple matches all messages arriving at LinkProcess:

(ANY -> LinkProcess, ANY)

Another example is two triples matching the opening of higher level connections in RIG (see Section 2).

(ANY -> ANY, Open)
(ANY -> ANY, Close)

If a higher resolution is required, a process may also explicitly declare a change in state by sending a pseudo-message. For example, LinkProcess declares a validity check of the input message.

(LinkIntInput-> LinkProcess, Input)
(LinkProcess -> System, Check)

Obtaining statistics of messages drastically reduce the trace data. However, the performance evaluator is still unable to understand the statistics. The main problem is a lack of the conceptual framework within which to evaluate these statistics. To provide the conceptual framework, I introduce the notion of a user activity- a sequence of messages implementing a given task. (A user transaction is another common term used in describing activities of a single user in a time sharing system [Watson 71]). The time intervals of a single message can then be expressed in terms of the time intervals of the user activities.

3.3.4 Statistics of a Sequence of Messages

There are two purposes for the abstraction of sequences of messages as a single activity: (1) concise description of long message traces and (2) establishment of a framework

within which to evaluate the time intervals of a single message. I define an elementary activity as a sequence of messages arriving at the same process; a composite activity is a collection of messages arriving at a group of processes.

First, I introduce the formalism for calculating the time intervals of an elementary activity. The formalism is also adequate for describing composite activities composed of processes that are running on the same processor. In the case of parallel processors, I extend the formalism with time intervals that measure the amount of overlapping.

To simplify the formalism here, I consider messages arriving at such a frequency that no pipelining occurs: the last message of an activity always occurs before the first message of the next activity. The performance evaluator describes an activity as a sequence of triples for which higher-level intervals are calculated. For example, two messages arrive at DisplayProcess: Input and Done.

```
ACTIVITY:  Display

  (LinkIntInput  ->LinkProcess, Input)
  (LinkIntOutput ->LinkProcess, Done)

END-ACTIVITY
```

The time intervals of the activity Display are defined as a sum of time intervals of individual messages.

```
Delay(Display)    = Delay(Command)    + Delay(Done)
Overhead(Display) = Overhead(Command)+ Overhead(Done)
Swapped(Display)  = Swapped(Command)  + Swapped(Done)
Interval(Display) = Interval(Command)
```

The Response time of the activity is the difference between the birth time of the first message and the completion of execution of the last message: the total time of the activity is the sum of execution time, system overhead and delay time.

```
Response(Display) = Finish(Done) - Birth(Command)

Total(Display) =
     Execution(Display)+Overhead(Display)+Delay(Display)
```

The activity is the conceptual framework for the sequence of messages at the lower-level. The response time of the activity measures the system at macro level. The difference between the measured response time Response(Display) and the

calculated total time Total(Display) provides a feedback to the performance evaluator on how well the set of measurements characterizes the time intervals of the activity.

Having found the correct set of messages that explain the time intervals of an activity, we proceed to evaluate the lower-level components. For example, if Delay(Display) constitutes a significant portion of Total(Display) time, the extensive CPU consumption by other processes is the bottleneck. To reduce the delay time, we increase the priorities of processes that are involved in this computation. In another example, computations of DisplayProcess is the bottleneck if Execution(Command) constitutes a significant portion of Execution(Display).

3.3.5 Time Intervals of Parallel Processes

In the case of parallel processes (which run on separate processors), we have to subtract the overlapped time of the processes' execution. In the example of distributed graphics, there are three independent processors: the CPU, the display device controller and the link device controller. For convenience, I repeat here the sequence of messages.

```
ACTIVITY:  Graphics

1)  (LinkIntInput  ->LinkProcess, Input)
2)  (LinkProcess   ->LinkIntOutput,Ack)
3)  (LinkProcess   ->DisplayProcess, Command)
4)  (DisplayProcess->DisplayInt,Draw)
5)  (LinkIntOutput ->LinkProcess, Done)
6)  (DisplayInt    ->DisplayProcess,Done)

END-ACTIVITY
```

Figure 8: Activity Graphics

Two messages are handled in parallel by the CPU and link output device (messages (2) and (3)). The link output handler accepts message Ack and starts the output operation on the link device. The next message Command is handled by DisplayProcess that is running on CPU. The amount of overlapping is calculated differently in the following three cases:

```
1) Overlap = 0,
             if Start(Command) > Finish(Ack)

2) Overlap = Finish(Ack) - Start(Command),
             if Finish(Command) > Finish(Ack)

3) Overlap = Finish(Command) - Start(Command),
             if Finish(Command) <  Finish(Ack)
```

Handling of messages is not overlapped in the first case, where the completion of the message Ack (the time stamp is Complete(Ack)) occurs before the beginning of the execution of the message Command (the time stamp is Start(Command)). Part of the handling of messages is overlapped in the second case, where message Ack is completed before message Command. Finally, the handling of messages is overlapped entirely in the third case, where message Command completes before message Ack.

Having estimated the total amount of overlapping, we calculate the total execution time by first summing up all the execution times of all the messages and subtracting from the total overlapped time. The remaining difference is the execution time of the activity. The following example presents computations for the activity DistributedGraphics:

```
Execution(DistributedGraphics) =

+  Execution  (LinkIntInput  ->LinkProcess, Input)
+  Execution  (LinkProcess   ->DisplayProcess, Command)
+  Execution  (LinkIntOutput ->LinkProcess, Done)
+  Execution  (DisplayInt    ->DisplayProcess,Done)
+
   (  Birth(DisplayInt    ->DisplayProcess,Done)
    - Complete(DisplayProcess->DisplayInt,Draw)
   )
+  (  Birth(LinkIntOutput -> LinkProcess,Done)
    - Complete(DisplayInt ->DisplayProcess,Done)
   )
```

3.3.6 Summary

For every message, I introduced a small set of intervals characterizing the processing time of the receiver (the interval is Execution), characterizing the ratio of incoming messages (Interval), and characterizing the delay time that the message spends in the input queue of the receiver (Delay).

To obtain statistics of messages, the performance evaluator described triples of messages consisting of the sender, receiver and message identifier. The same symbols were used both for programming of communicating processes and for describing finite state machine models of computations. Although the statistics of messages drastically reduced the amount of data, they were still difficult to understand for the performance evaluator. The main problem was a lack of the conceptual framework within which to evaluate the statistics.

To provide the conceptual framework, I defined a higher-level construct, a user activity representing a sequence of messages. An elementary activity is a sequence of messages arriving at the same process; a composite activity is a collection of messages arriving at a group of processes. To measure the amount of overlapping in composite activities, I introduced a new time interval, Overlap, that characterizes the amount of overlapping between parallel processes.

3.4 A Language for Describing Finite State Machines.

3.4.1 Motivation

So far, we have considered activities having a linear structure. This limitation is clearly unacceptable for processes that base their decisions both on the incoming messages and the internal state which is stored in local variables of the process.

For example, DisplayProcess may send the message Error back to the user instead of forwarding the message Draw to the display device. This occurs in the case where the user violates the protocol agreed upon during the initialization of the user.

(DisplayProcess -> LinkProcess, Error).

In both cases, the behavior of processes has changed due to the internal state of the process. The state changes are important events for the performance analysis of communicating processes.

3.4.2 Elementary Finite State Machines

This section uses finite state machine models to analyze the performance of communicating processes. Elementary finite state machines describe the behavior of a single process. Composite finite state machines describe the behavior of a

group of processes. The main advantage in using finite state machines is simplicity due to the total ordering of events in the context of the model.

One kind of event is the reception of a message by a process that is in the given state. In addition, the process can explicitly declare a change in state by sending a pseudo-message. An activity is a sequence of events occurring in a finite state machine passing from the initial state to the last state. From now on, I will use the term event for message and finite state machine for activity.

The example of distributed graphics requires four processes: LinkProcess, LinkIntOutput, DisplayProcess and DisplayInt. LinkProcess and DisplayProcess are each modeled with a simple finite state machine having two states: Idle and Busy. LinkIntOutput and DisplayInt (both are interrupt handlers) are each modeled with a finite state machine having only one state and one transition. The trace of messages is then converted to the trace of events (see below). Each event has a name of the finite state machine model, the current state label and the triple of the message.

```
LinkProcess.Idle:    (LinkIntInput  ->LinkProcess, Input)
LinkIntOutput.Idle:  (LinkProcess   ->LinkIntOutput,Ack)
DisplayProcess.Idle:(LinkProcess    ->DisplayProcess, Command)
DisplayInt.Idle:     (DisplayProcess->DisplayInt,Draw)
LinkProcess.Busy:    (LinkIntOutput ->LinkProcess, Done)
DisplayProcess.Busy:(DisplayInt     ->DisplayProcess,Done)
```

Unless otherwise specified, a sequence of transitions implies a sequence of state-transitions. The construct CONNECT breaks the sequence of state-transitions by explictly specifying the next state. A state having an alternate transition is defined by repeating the same state-label. For example, the state label "A" has two outgoing transitions: Input and Error. (A BNF definition of the language for describing finite state machines appears in Appendix B).

```
FSM: LinkProcess

A:  (LinkIntInput  ->LinkProcess, Input)
B:  (LinkIntOutput ->LinkProcess, Done)
    CONNECT(C)
A:  (LinkIntInput  ->LinkProcess, Error)
    ...
C:  ...

END-FSM
```

Figure 9: Elementary FSM, LinkProcess

State A has two outgoing transitions: the first to send message Done to LinkProcess and the second to send message Error. State B is followed by the new construct CONNECT(C) that specifies another entry to state C.

Composite finite state machines describe the execution of a group of processes that produce only a partially ordered collection of events. Some events that occur within different processes are still ordered in time due to the logic of computations. For example, communications between device handling processes and their interrupt handlers always occur in the same order.

Two processes communicate in full hand-shake if the first process sends a message to the second process and immediately waits for a reply from the second. In this case, the composite model simply describes the sequence of events. For example, LinkProcess and LinkIntOutput communicate in full hand-shake.

```
FSM: LinkComplete

  LinkIntOutput.Idle: (LinkProcess ->LinkIntOutput,Ack)
  LinkProcess.Busy: (LinkIntOutput ->LinkProcess, Done)

END-FSM
```

The sequence of events is further abstracted as a single event in the finite state machine at a higher-level (see below). I organize finite state machines in two levels of hierarchy: LinkComplete and Link. The higher-level model Link describes all events associated with the handling of an input packet.

```
FSM: Link
```

```
  (LinkIntInput  ->LinkProcess, Input)
  FSM(LinkComplete)

END-FSM
```

A new transition FSM(LinkComplete) occurs when the lower-level finite state machine (LinkComplete) passes from the initial state to the last state. Similarly, all events that are associated with the handling of display command are also described with two finite state machines: DisplayComplete and Display.

The next step for the performance evaluator is to describe a composite model of the entire system. Although some events may occur in arbitrary order, the composite model precisely defines different alternatives for the event ordering. In the composite model Graphics at state A, the two alternatives are either LinkComplete or DisplayComplete.

```
FSM: Graphics

        (LinkIntInput -> LinkProcess, Input)
        (LinkProcess  -> DisplayProcess, Command)
    A:  FSM(LinkComplete)
    B:  FSM(DisplayComplete)
        CONNECT(LASTSTATE)

    A:  FSM(DisplayComplete)
    C:  FSM(LinkComplete)

END-FSM
```

Figure 10: Composite Model, Graphics

Different alternatives within finite state machines are described uniformly either for a single process or for a group of processes.

3.4.3. Composite Models for Pipelined Computations

So far, we have considered only one activity in progress. This section deals with pipelined computations where activities overlap in time: the first event of a new activity occurs before the last event of the previous activity. (Recall that an activity is a sequence of events that occur in a finite state machine passing from the initial state to the last state).

Message traces with overlapped activities are difficult to analyze. Frequently, we find a few consecutive occurrences of the same type of message, each belonging to a different activity in progress. To retain the ability for calculating statistics of user activities, I extend the notion of an event to include the index of the activity in progress. A fully specified event has a finite state machine, an index of the current activity, a state label and a message triple.

For example, two consecutive occurrences of the message Input appear in the event trace as follows:

```
Graphics(i).A:   (LinkIntInput -> LinkProcess, Command)
Graphics(i+1).A: (LinkIntInput -> LinkProcess, Command)
```

Clearly, not everything can be described with this simplistic model. In addition to the statistics of each activity in progress, we would like to know how messages are distributed among processes. To answer this question, we need a composite model that describes a group of finite state machines, each modeling one activity in progress.

Pipelined computations produce very complicated traces of events due to the arbitrary ordering of events and activities in progress. To describe all possible cases in one composite model is impractical; instead, we consider only interesting cases that are selected by the user for the performance analysis of the computations. In addition to messages, I introduce a new type of event- a hardware interrupt.

```
FSM:  Display

A:  (LinkProcess    ->DisplayProcess, Command)
B:  (DisplayProcess->DisplayInt,Draw)
C:  (DisplayDevice ->Displayint,Interrupt)
D:  (DisplayInt     ->DisplayProcess,Done)

END-FSM
```

To describe a limited number of messages in progress, I introduce a new transition, INDEX [Figure 11]. A group of finite state machines change their roles to depict exactly these message in progress they are modeling by using the INDEX operation. A finite state machine with a label [i] after the INDEX transition becomes [i-1].

```
(i).A:   (LinkProcess    ->DisplayProcess, Command)
(i).B:   (DisplayProcess->DisplayInt,Draw)
(i).C:   (DisplayDevice ->Displayint,Interrupt)
(i).D:   (DisplayInt     ->DisplayProcess,Done)
(i+1).A: (LinkProcess    ->DisplayProcess, Command)
(i+1).B: (DisplayProcess->DisplayInt,Draw)
(i+1).C: (DisplayDevice ->Displayint,Interrupt)
(i+1).D: (DisplayInt     ->DisplayProcess,Done)
```

can be replaced with

```
(i).A: (LinkProcess    ->DisplayProcess, Command)
(i).B: (DisplayProcess->DisplayInt,Draw)
(i).C: (DisplayDevice ->Displayint,Interrupt)
(i).D: (DisplayInt     ->DisplayProcess,Done)
INDEX(Display)
(i).A: (LinkProcess    ->DisplayProcess, Command)
(i).B: (DisplayProcess->DisplayInt,Draw)
(i).C: (DisplayDevice ->Displayint,Interrupt)
(i).D: (DisplayInt     ->DisplayProcess,Done)
```

Figure 11: INDEX Operation in a Stream of messages

To describe global changes in the system, I introduce a new transition, PREDICATE, describing the exact system state as a vector of states of lower-level finite state machines. In the case of pipelined computations, we are interested in the vector of finite state machines modeling different activities in progress. The new transition, PREDICATE, helps to describe the vector of states [Figure 12].

```
(1)   PREDICATE(Display(i)=C, Display(i+1)=B)
(2)   Display(i).C:  (DisplayDevice -> DisplayInt, Interrupt)
(3)   Display(i+1).B:(DisplayProcess->DisplayInt,Draw)
(4)   PREDICATE(Display(i)=D, Display(i+1)=C)
(5)   Display(i).D:  (DisplayInt -> DisplayProcess, Done)
(6)   INDEX(Display)
```

Figure 12: PREDICATE Transition in Composite Models

The occurrence of transition (1) moves the models to the state where Display(i) is waiting for an interrupt (state C) and Display(i+1) is waiting for the device to become available (state B). The occurrence of Interrupt is immediately followed by the next command moving the model to another system state (4).

Although I introduced a repetitive pattern into the behavior of the system, the resulting finite state machine has many states. In particular, different load conditions result in different system states. A heavy load on the system results in only one message in progress:

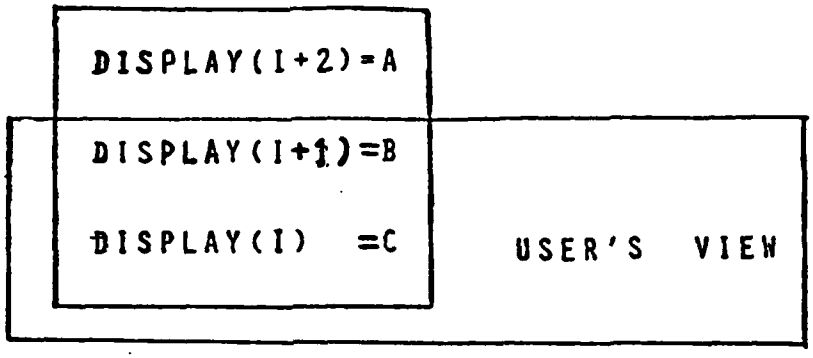
PREDICATE(Display(i)=C, Display(i+1)=A)

A light load on the system results in many messages in progress (if the remote process can generate data faster than the graphics device can display).

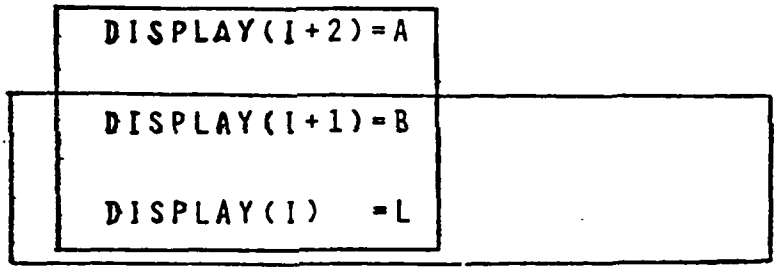
PREDICATE(Display(i)=C, Display(i+1)=B, Display(i+2)=B, ...)

Modeling all system states is impractical due to the large number of them. An alternative is to describe a limited system state- a user view of computations in the system [Figure 13]. Three messages in progress are each described by a finite state machine. The user is concerned with two models: Display[i] that is in state B and Display[i+1] that is in state C. The hardware interrupt moves model Display[i] to state L. At this point, the user applies INDEX operation to consider the next pair of finite state machines.

I have already developed a formalism to describe the user view of computations in the system: the sequence of transitions from (1) to (5) describes the system state for only two messages in progress although in reality there might be many more messages in progress.



DISPLAY(I).INTERRUPT



INDEX(DISPLAY)

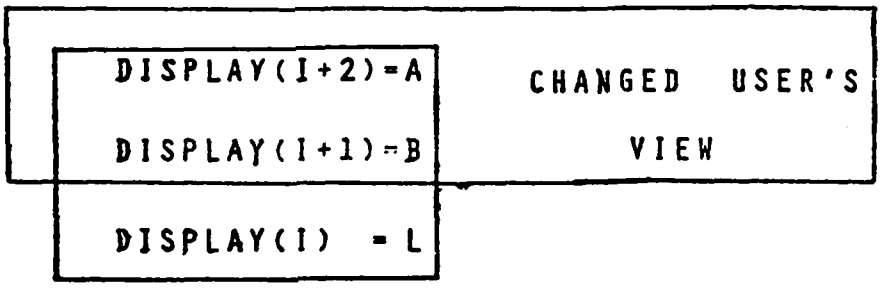


Figure 13: User Windows

To ensure completeness of the model, I introduce a special RESET state. All states that are entered with a PREDICATE transition are connected to RESET. If none of the specified PREDICATE transitions occur, the model enters RESET state. If the finite state machine model is not accurate in that it does not capture system states occurring often in the execution of the system, most statistics are collected within the RESET state. Exit conditions from the RESET state are defined with PREDICATE transitions.

The sequence of five transitions described above constitutes an ideal loop of always having one command in progress. However, after an INDEX operation we may encounter a state with no messages in progress.

```
INDEX(Display)
PREDICATE(Display(i)=C, Display(i+1)=A)
Display(i).C:  (DisplayDevice -> DisplayInt, Interrupt)
Display(i).D:  (DisplayInt -> DisplayProcess, Done)
PREDICATE(Display(i)=0, Display(i+1)=A)
```

If the above sequence of transitions occurs, the display device becomes idle: Either the remote process has not sent the data or LinkProcess was not able to handle it. To discover the reason, we apply transition PREDICATE to an entirely different model for LinkProcess.

```
PREDICATE(LinkProcess(i)=A)
```

If the transition occurs, LinkProcess has not received message (i), pointing out the foreign process- the bottleneck of the system. If the transition does not occur, LinkProcess has seen the message but for some reason has not delivered it to DisplayProcess. Then, we might want to describe a more detailed model of the computation. The transition PREDICATE allows us to define an arbitrary system state as a vector state of selected finite state machines.

3.5 Summary

This chapter described a formalism for performance analysis of communicating processes. First, it described the basic properties of messages traces and introduced three time stamps: Birth, Start, and Finish. On the basis of those intervals, for every message the system calculates three time intervals: Execution, Interval, and Delay. Analysis of those measurements was still difficult due to the lack of a conceptual framework within which to evaluate those statistics. Then, I introduced the notion of an activity as a collection of messages serving a single user request. Elementary activities represent a sequence of messages arriving at the same process; composite activities represent a collection of messages arriving at a group of

processes. Analogous intervals were defined to describe the performance of elementary activities. In the case of composite activities, I extended the formalism with the new interval, Overlap, that characterized the amount of overlapping between parallel processes.

Further, I introduced a finite state machine model describing the semantics of the message traces. The time intervals that were used to describe messages were extended to describe events defined by state transitions in the finite state machine model. The activity is defined as a sequence of events occurring in a finite state machine passing from the initial state to the last state. Elementary activities were described by elementary finite state machines, and composite activities by composite finite state machines. To reduce the number of states in the composite models, I introduced three new transitions: (1) FSM describes a long sequence of messages, (2) PREDICATE describes the exact system state as a vector of process states, and (3) INDEX describes a limited number of messages in a stream. The suitability of the new transitions was tested by real measurements.

Although composite models may have a large number of states, in our experience with RIG only a small number of states are actually reached during the system's execution. To find those states, howver, is very difficult and requires a deep understanding of the system. Another difficulty is in programming of finite state machines in a symbolic language similar to that described in this chapter. A drawing of a finite state machine is a better representation allowing one to immediately grasp various alternate transitions in the model.

The growing interest in message-based computing and in formal description of communicating processes suggests that many future systems should be implemented or at least designed using finite state machines ([Estrin et al., 78] and [Riddle et al., 78]). In those cases, the performance evaluator will immediately have accurate finite state machine models available for performance analysis. The value of such a facility is another good reason to use finite state machines in the design process.

4. Examples

4.1 Introduction

This chapter describes how different finite state machines are formulated and applied to the analysis of RIG. The obtained results demonstrate the value of finite state machine for performance analysis of communicating processes.

The chapter is based on two examples from the RIG system: the virtual terminal and the file system. Section 4.2 deals with large scale computations. In RIG, the initialization of a terminal is such an example. Section 4.3 deals with pipelined computations. The example is a user program writing to a sequential file. This entails pipelined computations because the CPU operations are overlapped with the disk operations.

This chapter uses the formalism introduced in Chapter 3 to present results obtained from the analysis of RIG. Chapter 6 describes how the same finite state machine formalism is applied to two different areas: validation of reliable transmission protocols and efficient implementation of higher-level protocols. These two examples further support the possition that finite state machines are valuable and practical models of communicating processes for the purpose of design, implementation and performance evaluation.

4.2 Large Scale Computations

4.2.1 Introduction

Here we are concerned with long sequences of messages produced by many processes. Processes communicate in full hand-shake: a process sends a message and immediately waits for reply. Although this represents an extreme case that reduces computations of a potentially parallel program to a sequential program, this kind of computation is frequently found in various initialization scenarios of multi-process systems. In RIG, the initialization of a terminal is characterized by a full-handshake style of communication [Lantz et al., 79]. Overall four hundred messages are passed among fifteen processes; out of them eight processes are started [Gertner 79b]. All computations are performed on a single processor. Reading this section below should help explain why the initialization of a terminal is so complicated.

The section has four major subsections: Subsection 4.2.2.

contains a fragment of a message trace pointing out the problems in modeling a large number of processes. Next three subsections describe finite state machines for analyzing these traces. Subsection 4.2.3 begins with a simple model for a user program reading a block of data from the disk. Subsection 4.2.4 uses the simple model to describe a higher-level finite state machine for reading a file of data. Subsection 4.2.5 describes the highest-level finite state machine model for initializing a terminal in RIG.

4.2.2 Message Trace

This section points out difficulties in analyzing a long sequence of messages. The example is a fragment of the message trace required to initialize a terminal. The format of the message trace is slightly modified to improve readability. For the purpose of presentation, the message trace is divided into eight parts:

1) The system is idle; only clock interrupt messages periodically appear in the system.

2) A user hits a "return" key causing the terminal to initialize. The message DCUInterrupt indicates this event.

3) The ResourceManager process requests ProcessManager to start a new process (message CreateProcessMsg). The Process Manager is responsible for reading in the process definition tables and parsing them. Overall, twenty messages occur by the time the new process (InitMonitor) is created. The ProcessManager returns the process identifier to the ResourceManager (message PmReply: 115).

Here, one problem for the performance evaluator is to select messages of interest in a long sequence. Some messages always occur in the system independent of the application. For example, one second may expire and the Timer process is notified. Moreover, this time a five second interval expires causing the Timer to notify the TenServer process with the message FiveSecInterrupt. A finite state machine having state-transitions each defined as reception of a message is extremely valuable model for selecting messages of interest.

The large number of messages arriving at the FileSystem process make it difficult to concentrate on the main issue, the creation of new processes. (Recall that we consider only a fraction of the entire message trace required to initialize a terminal). In this case, I will use a hierarchy of finite state machines to concisely describe the long sequence of messages. A lower-level finite state machine describes all messages arriving at the FileSystem.

A higher-level finite state machine contains a single transition defined as a sequence of messages causing the lower-level finite state machine to pass from the initial state to the last state.

4) The ProcessManager requests NewProc (a system process responsible for creating a process map as required by the Eclipse hardware manual). The newly created InitMonitor sends a request to start the Monitor process. Two messages CreateProcessMsg and PmReply appear in arbitrary order. In this particular case, the ProcessManager receives the message CreateProcessMsg before ResourceManager receives the message PmReply. This particular ordering is random, requiring the finite state machine model to account for all the possible alternatives.

5) Another process is created (Monitor=115). Here, ProcessManager replies to InitMonitor (PmReply: 116) before the newly created process (Monitor) sends any message.

6) The StatusServer process is also created (StatusServer=117).

7) The ResourceManager requests arguments from InitMonitor. Although at this point there are two messages outstanding for InitMonitor (PmReply: 117, and RequestArgsMsg), it always receives these two messages in the same order. This is because InitMonitor communicates in a full hand-shake style with ProcessManager. This example demonstrates how the full hand-shake style helps to describe a finite state machine having a single path modeling the receipt of two messages. In part seven of the message fragment, all messages are related to the initialization of a terminal but they are not related to the creation of new processes.

8) The LineHandler process is created (LineHandler=120).

```
1)
 System           -> Timer ,            ClockInterrupt
 System           -> Timer ,            ClockInterrupt

2)
 System           -> TerminalInput,     DCUInterrupt
 System           -> Timer ,            ClockInterrupt
 TerminalInput    -> ResourceManager ,  DataMsg
 ResourceManager  -> TerminalOutput,    LineMsg

3)
 ResourceManager  -> ProcessManager ,   CreateProcessMsg
 ProcessManager   -> FileSystem ,       OpenMsg
 System           -> TerminalOutput,    DCUInterrupt
 System           -> FileSystem ,       DiskInterrupt
 System           -> FileSystem ,       DiskInterrupt
 System           -> Timer ,            ClockInterrupt
 Timer            -> TenServerProcess , FiveSecInterrupt
 System           -> FileSystem ,       DiskInterrupt
 System           -> FileSystem ,       DiskInterrupt
 FileSystem       -> ProcessManager ,   FileReply
 ProcessManager   -> FileSystem ,       InputMsg
 System           -> FileSystem ,       DiskInterrupt
 FileSystem       -> ProcessManager ,   OutputMsg
 ProcessManager   -> FileSystem ,       CloseMsg
 System           -> FileSystem ,       DiskInterrupt
 FileSystem       -> ProcessManager ,   CloseReply

4)
 ProcessManager   -> NewProc ,          NewProcCreate
 Init-Monitor     -> ProcessManager ,   CreateProcessMsg
 ProcessManager   -> FileSystem ,       OpenMsg
 ProcessManager   -> ResourceManager ,  PmReply  : 115
//InitMonitor = 115

5)
 System           -> TerminalInput,     DCUInterrupt
 TerminalInput    -> TerminalInput,     ErrorMsg
 System           -> FileSystem ,       DiskInterrupt
 System           -> Timer ,            ClockInterrupt
 System           -> FileSystem ,       DiskInterrupt
 System           -> FileSystem ,       DiskInterrupt
 FileSystem       -> ProcessManager ,   FileReply
 ProcessManager   -> FileSystem ,       InputMsg
 System           -> FileSystem ,       DiskInterrupt
 FileSystem       -> ProcessManager ,   OutputMsg
 ProcessManager   -> FileSystem ,       CloseMsg
 System           -> FileSystem ,       DiskInterrupt
 FileSystem       -> ProcessManager ,   CloseReply
 ProcessManager   -> NewProc ,          NewProcCreate
 ProcessManager   -> Init-Monitor,      PmReply  : 116
//Monitor = 116
```

```
6)
 Init-Monitor     -> ProcessManager ,   CreateProcessMsg
 ProcessManager   -> FileSystem ,       OpenMsg
 System           -> FileSystem ,       DiskInterrupt
 System           -> FileSystem ,       DiskInterrupt
 System           -> FileSystem ,       DiskInterrupt
 System           -> FileSystem ,       DiskInterrupt
 System           -> FileSystem ,       DiskInterrupt
 System           -> FileSystem ,       DiskInterrupt
 FileSystem       -> ProcessManager ,   FileReply
 ProcessManager   -> FileSystem ,       InputMsg
 System           -> FileSystem ,       DiskInterrupt
 FileSystem       -> ProcessManager ,   OutputMsg
 ProcessManager   -> FileSystem ,       CloseMsg
 System           -> Timer ,            ClockInterrupt
 System           -> FileSystem ,       DiskInterrupt
 FileSystem       -> ProcessManager ,   CloseReply
 ProcessManager   -> NewProc ,          NewProcCreate
 ProcessManager   -> Init-Monitor,      PmReply : 117
//StatusServer = 117

7)
 ResourceManager -> Init-Monitor,      RequestArgsMsg
 Init-Monitor    -> ResourceManager ,  ProcessArgsMsgReply
 Init-Monitor    -> ScreenHandler ,    OpenMsg
 ScreenHandler   -> TerminalOutput,    InitScreenMsg
 System          -> TerminalOutput,    DCUInterrupt
 ScreenHandler   -> TerminalOutput,    GetTerminalProfile
 TerminalOutput  -> ScreenHandler ,    InitScreenMsg
 ScreenHandler   -> Init-Monitor,      OpenReply

8)
 Init-Monitor     -> ProcessManager ,   CreateProcessMsg
 ProcessManager   -> FileSystem ,       OpenMsg
 System           -> Timer ,            ClockInterrupt
 System           -> FileSystem ,       DiskInterrupt
 System           -> FileSystem ,       DiskInterrupt
 System           -> FileSystem ,       DiskInterrupt
 FileSystem       -> ProcessManager ,   FileReply
 ProcessManager   -> FileSystem ,       InputMsg
 System           -> FileSystem ,       DiskInterrupt
 FileSystem       -> ProcessManager ,   OutputMsg
 ProcessManager   -> FileSystem ,       CloseMsg
 System           -> FileSystem ,       DiskInterrupt
 FileSystem       -> ProcessManager ,   CloseReply
 ProcessManager   -> NewProc ,          NewProcCreate
 ProcessManager   -> Init-Monitor,      PmReply
//LineHandler = 120
```

In summary, initializing a terminal produces a long sequence of messages. All messages depicted above, constitute only a fragment of all messages required to initialize a terminal. Three difficulties encountered by the performance evaluator are: 1) a large number of communicating processes; 2) a collection of messages that are not related to the main line of computation, creation of new processes; 3) a large number of messages arriving at the FileSystem.

The next three sections describe one possible solution to these problems. To analyze messages arriving at the file system, I use a hierarchy of finite state machines. A higher-level model contains a transition modeling a sequence of messages required to read an entire file. Messages that are not related to creation of new processes are not included in the composite model of the system. The error is estimated by accumulating statistics for all these messages. The full hand-shake style of communication between processes makes it possible for the performance evaluator to describe the expected sequence of messages and subsequently to encode them in a finite state machine model.

4.2.3 Reading a Block of Data

This section describes an example of a user reading a block of data from the disk, ignoring for the moment the issues of multiplexing and pipelining (those problems will be addressed in Section 4.3). I begin with two elementary finite state machines: one for the file system and the other for the disk handler. Then I describe a composite model for both processes. In the presentation, I suggest a methodology to describe finite state machines and how to use them.

A user program ("the User") reading a block of data from the disk requires services from the file system ("the FileSystem") and the disk device ("the DiskHandler"). The User and FileSystem communicate with messages ReadBlock and ReadDone, the FileSystem and DiskHandler with DiskCommand and DiskDone [Figure 14].

It is convenient to model the DiskHandler with only one state and one transition. Having received the message DiskCommand, DiskHandler starts the device (a long transition). The disk interrupt moves the model back to the initial state.

The FileSystem has three states and four transitions. It starts in state A, and upon receiving message ReadBlock from the User, moves to the state B, a decision state. In state B, the FileSystem decides whether to read directory blocks or not. If needed, the FileSystem moves to state C, by issuing DiskCommand to read the directory; otherwise, the

FileSystem remains in state B by issuing DiskCommand to read the user data. The same message identifier is used in either case: whether reading the user data or the directory. The finite state machine models help to identify the meaning of these messages using state information. For the purpose of performance measurements, we can exclude from the model the decision of the FileSystem to remain in state B.

To encode these models, I have used both documentation and actual code of the file system. The documentation that describes interprocess communications provides information sufficient to describe three transitions: ReadBlock, DiskDone, and EOF (End Of File). Further, I have examined the code of the file system and found a state that requires several disk operations for updating user directories. The code has been modified to declare a change in the internal state for the purpose of performance monitoring. Although the code dealing with directories is complicated, I approximate the entire computation with only two states: B and C [Figure 15]. The accuracy of the model is sacrificed for simplicity. The simplified model still retains states and transitions that are important for performance analysis. The transition ReadDirectory followed by DiskDone accumulates statistics for all directory operations.

In addition to the notation described in Section 2, I use drawings of finite state machines. Similar drawings have been used to describe SNA protocols [IBM SNA]. In the graphical notation, each state is indicated by a vertical line named either at the top or the bottom. The vertical lines have circles with incoming or outgoing arrows. Each transition between states is represented by a horizontal arrow with the following properties:

* The tail of the arrow starts at a circle on the state line corresponding to the initial state (before the transition).

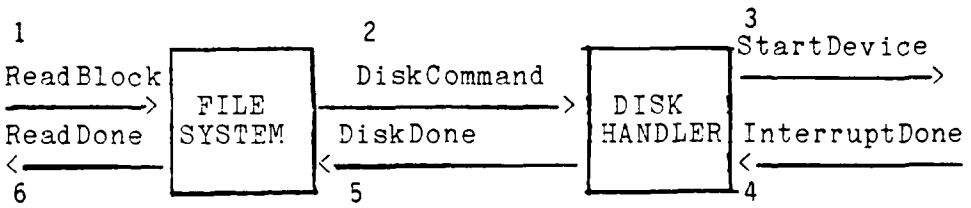
* The head of the arrow ends at a circle on the state line corresponding to the next state for the given transition.

* The activities or the output of the transition appear as comments directly below the transition line.

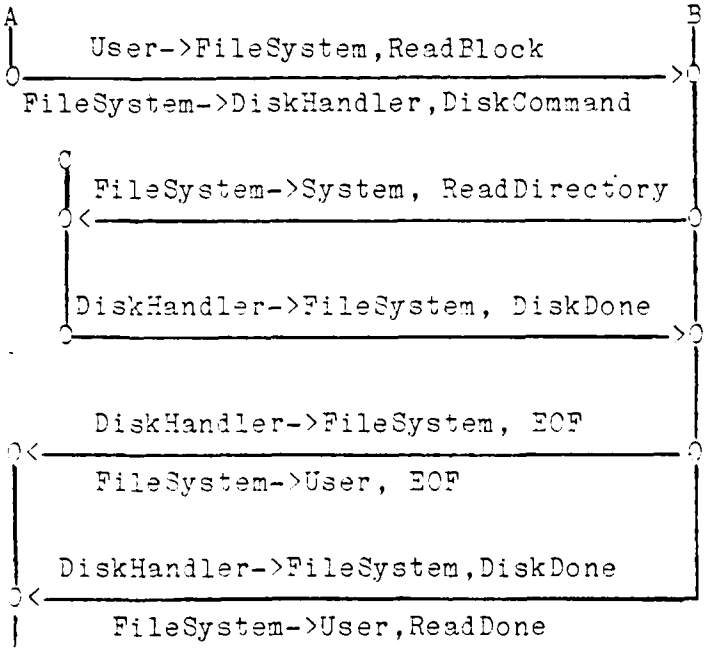
* The input associated with the transition or the logical condition causing the occurrence of the transition appears directly above the state transition line.

* The transition arrow might represent a loop causing the finite state machine model to stay in the same state.

Block Diagram, Read Block



FSM Graph, FileSystem



FSM Graph, DiskHandler

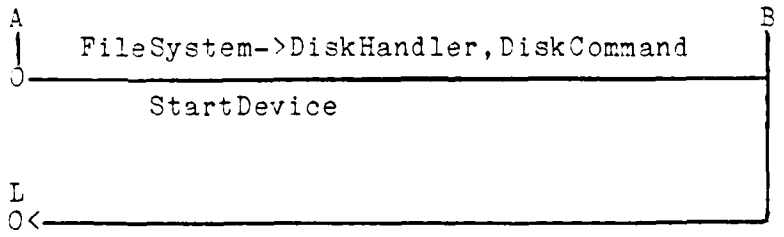


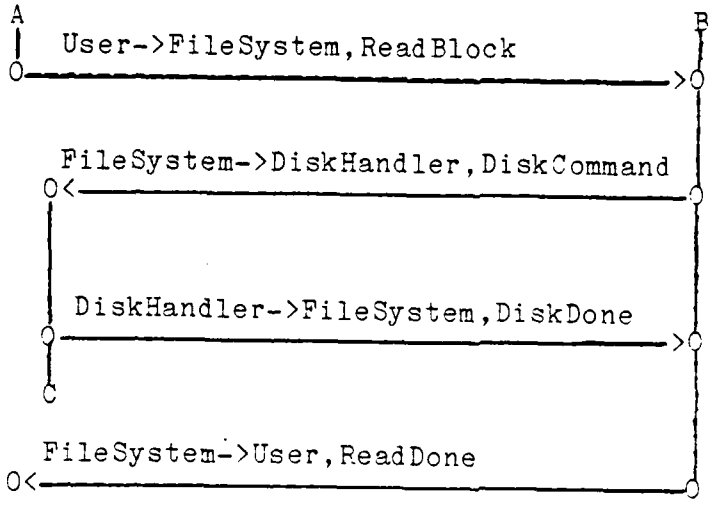
Figure 14: Read Block

Until now, we considered two finite state machines: one for the file system and one for the disk handler. The next step is to build a composite model capturing the behavior of both processes. There are two reasons for doing this: (1) the composite model can later be used as a single transition in higher-level models; and (2) statistics of a single model are better understood by the performance evaluator who views the entire system as a sequence of events in terms of the composite model.

The description of the composite model ReadBlock is easy due to the full hand-shake in the interprocess communication. Since only one finite state machine is active at a time, the composite model is obtained by simply specifying states from different finite state machines in the order of their appearance. Surprisingly, in RIG, composite models are easier to describe than elementary models. This is because all changes in the system state are reflected in the trace of messages. The number of states in the composite model is small due to the very conservative style in the use of messages. (In RIG, there are basically two styles of communications: full hand-shake and message streaming. Until now, we considered only full hand-shake; Section 4.3 considers message streaming.)

To further simplify the composite model, I use a single transition for reading a block of data that contains either user data or directory information. The final version of the finite state machine has only three states: A, B, and C [Figure 15].

FSM Graph, Read Block



FSM Program, Read Block

```
FSM: ReadBlock
          (User->FileSystem,ReadBlock)
B:        (FileSystem ->DiskHandler,DiskCommand)
          (DiskHandler->FileSystem,DiskDone)
          CONNECT(B)
B:        (FileSystem ->User,ReadDone)
END-FSM
```

Figure 15: FSM Read Block, Simplified Composite Model

In summary, I described model ReadBlock to be used as a single transition in the higher-level models of reading a file (Section 4.2.3). I began with two elementary finite state machines: one describing the file system and another the disk handler. Describing the file system was difficult due to the directory operations that required analysis of the program code. Composite models were easier to describe because all state changes in the system were expressed in messages. The description of the composite model was further simplified by leaving only those transitions that were important for performance analysis.

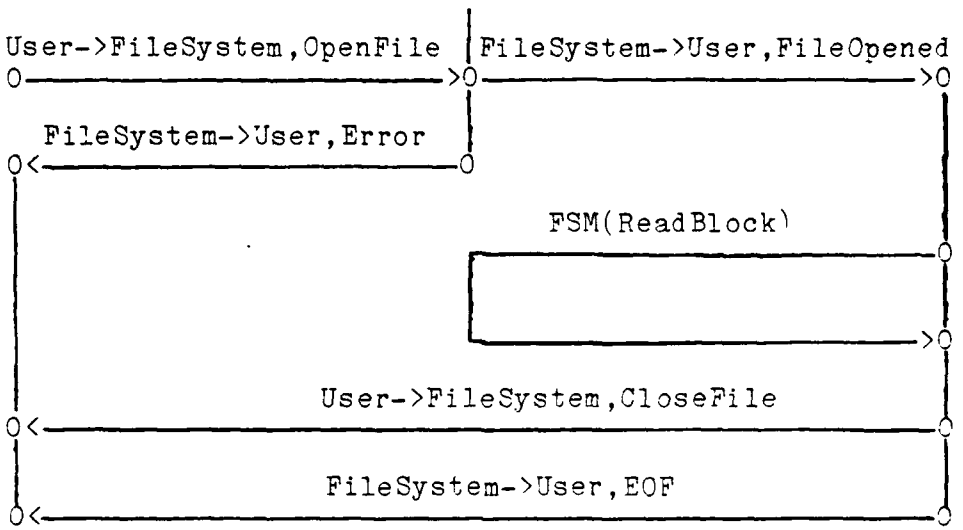
4.2.4 Reading a File

This section uses the finite state machine model for reading a block of data to describe a higher-level finite state machine for reading a file. At the higher-level, the transition

FSM(ReadBlock)

models the occurrence of a sequence of events that take place as a finite state machine passes from the initial state to the last state. The finite state machine ReadFile starts in "idle state" A, and by receiving the message OpenFile from the User moves to state B, a decision state. In the case of an error message (the file does not exists) the FileSystem replies to User either with an error (for example, the file does not exist) and moves back to state A, or with FileOpened and moves to C. In state C, the FileSystem continues reading blocks (the abstract transition is FSM(ReadBlock)) until the end of the file (EOF message) and closes the file (CloseFile) at the user's request. Again, composite models were easier to describe than elementary models.

FSM Graph, Read File



FSM Program, Read File

```
FSM: ReadFile
          (User->FileSystem,OpenFile)
B:        (FileSystem ->User,Error)
          CONNECT(LASTSTATE)
B:        (FileSystem ->User,FileOpened)
C:        FSM(ReadBlock)
          CONNECT(C
C:        (User->FileSystem,CloseFile)
          CONNECT(LASTSTATE)
C:        (FileSystem ->User,EOF)
          CONNECT(LASTSTATE)
END-FSM
```

Figure 16: FSM Read File

4.2.5 Initializing a Terminal

This section describes the highest-level finite state machine modeling the initialization of a terminal in RIG. Lower-level models that are described in previous sections are used as transitions. Although processes communicate in a full hand-shake style, the composite model is still difficult to describe due to the large number of processes. The difficulty is in knowing the exact order of computation of so many processes.

To represent all possible alternatives is impractical. Many branches will complicate the model without contributing to a better understanding of the performance. In this case, I make no attempt to describe a composite model for the entire system. Instead, I describe the composite model of the major subsystem. The documentation of the RIG system has been sufficient for me to acquire the knowledge necessary to describe the model. This is an example of what one should be able to do with adequate system documentation.

Initializing a terminal requires eight new processes: InitMonitor, responsible for creating all the processes handling the terminal, LineHandler, responsible for multiplexing the physical line, and three pairs of processes Monitor and PAD-Monitor, StatusServer and PAD-StatusServer, Executive and PAD-Executive. Each pair of processes is responsible for creating and destroying different kinds of terminal windows (regions). Monitor is responsible for creating new user regions of the screen. StatusServer is responsible for controlling the status of all currently active regions. Executive is responsible for handling of user requests.

In the finite state machine description, the first transition models the creation of process InitMonitor [Figure 17].

InitMonitor = FSM(CreateProcess)

To measure the overhead in exchanging arguments among processes and opening logical connections, I used two finite state machines: OpenConnection and PassArguments. Statistics of those models, together with statistics of the composite model, constitute time intervals of the overall system. The experiment [Figure 17] begins with the message

(ResourcesManager -> InputProcess, Input)

being sent to the ResourceManager responsible for handling physical lines; the experiment ends with the message

(Executive -> LineHandler, LineEdit)

received by the LineHandler responsible for multiplexing a physical line.

The transition FSM (TerminalComponents) models the creation of LineHandler and two virtual screens: Monitor and StatusServer. The order in which processes are created is unimportant; therefore, I use a finite state machine of type COLLECT that models a collection of messages arriving in arbitrary order.

```
FSM:      CreateMonitor

          Monitor     = FSM(CreateProcess)
          Pad-Monitor= FSM(CreateProcess)

          ACTIVATE(Monitor-Output)
END-FSM

FSM:      MonitorOutput

          (Monitor -> Monitor-Pad, ANY)
          (Monitor-Pad -> Monitor, PadReply)

END-FSM

FSM:      CreateStatusServer

          StatusServer = FSM(CreateProcess)
          Pad-StatusServer= FSM(CreateProcess)

          ACTIVATE(StatusServer-Output)
END-FSM

FSM:      StatusServerOutput

          (StatusServer -> StatusServer-Pad, ANY)
          (StatusServer-Pad -> StatusServer, PadReply)

END-FSM


FSM(COLLECT):   TerminalComponents

          FSM(CreateMonitor)
          FSM(CreateStatusServer)
          LineHandler = FSM(Createprocess)
END-FSM

FSM:      PassArguments

          (ANY -> ANY, RequestArgMsg)
          (ANY -> ANY, ProcessArgsMsgReply)
END-FSM

FSM:      OpenConnection

          (ANY -> ANY, OpenMsg)
          (ANY -> ANY, OpenReply)
END-FSM
```

Figure 17: FSM Initialize Terminal
(continued on the next page)

```
FSM:  CreateProcess

          Requester =(ANY -> ProcessManager, CreateProcessMsg)
          FSM(ReadFile)
          (ProcessManager -> MapCreator, NewProcCreate)
          (MapCreator     -> ProcessManager, MapCreated)
A:        (ProcessManager -> Requester, PMReply)
          (NewProcess     -> System,  FirstScheduling)
          CONNECT(LASTSTATE)
A:
          (NewProcess     -> System,  FirstScheduling)
          (ProcessManager -> Requester, PMReply)

END-FSM


FSM:  InitializeTerminal

   (Resourcesmanager -> TerminalInput, InputMsg)
   InitMonitor = FSM(CreateProcess)

   ACTIVATE(PassArguments)
   ACTIVATE(OpenConnection)
   ACTIVATE(ScreenManagement)

   FSM(TerminalComponents)

   (MapCreator->ProcessManager, ProcessDied)
   Executive     = FSM(CreateProcess)
   Pad-Executive= FSM(CreateProcess)

   ACTIVATE(Executive-Output)
   (Executive -> LineHandler, LineEdit)

END-FSM
```

Figure 17: FSM Initialize Terminal
(continued)

The statistics produced by this model point out that transition FSM(ReadFile) consumes about 25% of the total time required to initialize a terminal. By keeping process definition tables in memory, we could speed up the terminal initialization by 25%. Although many messages are required to establish logical connections, the system spends only 10% of its time in modules FSM(PassArguments) and FSM(OpenConnection). Therefore, the performance of the system was not severely affected by requiring all processes to conform to those conventions. The statistics of those transitions are as follows:

```
Statistics-FSM:  ReadFileEvents
  NumSamples:=   106  Execution:=   2608 Overhead:=  600
  Swapped:=    14     IdleWaitDevice:=   1312

Statistics-FSM:  OpenConnection
  NumSamples:=   18   Execution:=   310  Overhead:=  113

Statistics-FSM:  PassArguments
  NumSamples:=   8    Execution:=   150  Overhead:=  50
```

4.2.6 Results

This section considered long sequences of messages produced by many processes communicating in full hand-shake. The example was the initialization of a terminal in RIG. Overall four hundred messages were passed among fifteen processes. To describe such a long sequence of messages, I introduced a hierarchy of four levels:

a) Level 1, FSM(ReadBlock) modeled computations of the file system and disk handler in reading a block of data.

b) Level 2, FSM(ReadFile) modeled computations for the file system and user in reading a file.

c) Level 3, FSM(CreateProcess) modeled computations of four processes: ProcessManager, MapCreator, FileSystem and DiskHandler.

d) The highest-level 4, FSM(StartTerminal) modeled all fifteen processes exchanging four hundred messages.

This hierarchy allows the performance evaluator to express the statistics of very detailed computations of the file system in terms of the statistics of initialization of a terminal in RIG. This suggested a more efficient initialization of a terminal in RIG by keeping the processes' definition tables in memory, thereby avoiding the file system accesses entirely. This kind of information was gathered naturally by using finite state machines.

This section considered a long sequence of messages produced by many processes communicating in a full hand-shake style. This kind of computation is also found in initialization of some other systems. For example, the initialization of RUNTOOL in the NSW system involves twelve distinct processes and well over forty process activations ([NSW 77] and [Schantz 79]). The methodology developed in this section can be applied to the analysis of the NSW system. Although processes run on distinct computers, most of the time they communicate in full hand-shake, thereby making it possible to describe a simple finite state machine modeling the behavior of the entire system.

The NSW (National Software Works) is a software system that provides uniform accesses to diverse computers (hosts) in a network. It facilitates the use of a wide variety of software tools. Specific knowledge about the location of the tool or its particular environment is often not needed.

4.3 Pipelined Computations

4.3.1 Introduction

This section is concerned with pipelined computations described by repetition of sequences of messages (user activities). A new activity frequently begins before the previous activity has been completed; therefore, the handling of some messages is overlapped in time. The example used in this section is a user streaming data to a disk. A separate disk controller allows the overlap of disk accesses with CPU computations. I demonstrate how the knowledge of the system helped to identify a small number of states in the composite model.

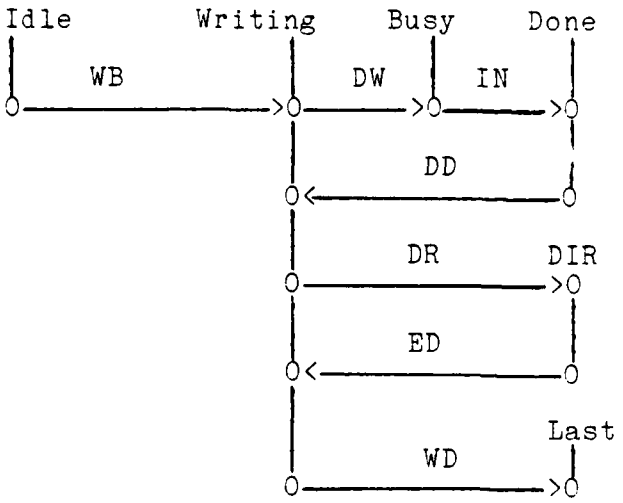
This section has three major subsections: Subsection 4.3.2 describes an elementary finite state machine, modeling a user that writes a block of data. Subsection 4.3.3 describes the composite finite state machine, modeling two activities in progress. Subsection 4.3.4 describes a composite finite state machine modeling two users. Different finite state machines were formulated and applied to the same data to extract different kinds of information.

4.3.2 Writing a Block of Data

The first step is to describe a simple finite state machine that models only one message in progress and has six states: Idle, Writing, Busy, Done, Directory and Last [Figure 18]. The model is very similar to the ReadBlock model that is described in Section 4.2.2. To represent more accurately the overlap between the CPU and disk controller, I use interrupts as events.

Statistics of this simple model revealed that 40% of all the disk operations are performed to update the directories. In the system with a single user, the disk controller requires, on average, 50 milliseconds to complete a single operation. Every second 20 disk operations are completed: 13 for the user and 7 for the directories. Consequently, the average interval between user commands is at least 77 milliseconds. During directory operations, the file system receive about 5 commands that keep the disk controller busy. When one specific command is in progress (Write[i]=Busy), the CPU can be used to perform other computations. To perform the next command, the file system spends 15 milliseconds: 10 milliseconds to handle the user's request (the transitions is WB) and 5 milliseconds to handle the interrupt message (DD). This simple model points out that the directory operation is the bottleneck of the system.

FSM Graph, Write Block



Abbreviations:

```
WB   - (UserProcess->FileSystem, WriteBlock)
DW   - (FileSystem ->DiskHandler,DiskWrite)
DD   - (DiskHandler->FileSystem, DiskDone)
IN   - (Disk       ->DiskHandler,Interrupt)
WD   - (FileSystem ->System,     WriteDone)
DR   - (FileSystem ->System,     DirectoryOperation)
ED   - (FileSystem ->System,     EndDirectory)
```

Figure 18: Write Block

4.3.3 Composite Model for Two Messages in Progress

To measure the overlap between the CPU and disk operations we need a composite model describing several messages in progress. To simplify the description of the composite model, I consider only two messages in progress. This is sufficient to monitor one message being handled by the disk controller and the next message being prepared by the CPU.

To further reduce the number of states in the composite model, I describe only those states that are most likely to occur. The following properties (or global state-transitions) of the system help to reduce the number of states. The description of each property is followed by a fragment of system transitions used later in the composite model.

1) Start(Write[i].DD) < Start(Write[i+1].IN

An interrupt message is always received before the next interrupt occurs. In the composite model, the following three transitions occur in the same order:

```
PREDICATE(Write[i+1]=Busy), Write[i]=Done)
Write[i].(DiskHandler -> FileSystem, DiskDone)
PREDICATE(Write[i+1]=Busy,  Write[i]=Writing)
```

2) Start(Write[i].DD < Start[Write[i+2].WB

The file system receives an interrupt message with priority. The following three transitions appear in the composite model:

```
PREDICATE(Write[I+1]=Idle), Write[i]=Done)
(DiskHandler -> FileSystem, DiskDone)
(User  -> FileSystem, WriteBlock)
```

3) PREDICATE(Write[i]=Busy), not Write[i+1]=Busy])

The disk device handles at most one command at a time. The following three transitions appear in the composite model:

```
PREDICATE(Write[i]=Writing, Write[i]=Busy)
Write[i].(Disk -> DiskHandler, Interrupt)
(FileSystem -> DiskHandler, WriteBlock)
```

4) Upon completion of the operation, the DiskHandler immediately fetches the next command. The following three transitions appear in the composite model:

```
PREDICATE(Write[ i+1 ]=Writing, Write[ i ]=Busy)
Write[ i ].(System->DiskHandler, Interrupt)
Write[ i+1 ].(FileSystem->DiskHandler, DiskWrite)
```

5) At most two commands in progress are considered. (A user defines predicates for two commands in progress).

The composite finite state machine [Figure 19] starts in "no write request" state A, in "one write request" state B, or in "two write requests" state C. In state A, the first write request moves (after a long delay) to state B (the transition is (O, WB)). Now, a new request moves to state C (the transition is (WB, WB)) or the disk command moves to "one write in progress and there are no more user requests" state D. But, in state D, the finite state machine can still "catch up" by receiving a "next write request" (the transition is (WR, DW)) and moves to state E.

In the decision state B, the FileSystem may decide to suspend temporary user requests and engage in updating the system's directory. Computations without the overlap between the CPU and disk are described by five states: A, B, D, N, and P. The remaining seven states describe the overlapped computations: D, E, F, G, H, and back to D or E.

In state E, one message is being handled by the disk controller and another is ready for execution. If another user message arrives, the model accumulates statistics in state E (the loop with the transitions WD). In this case, there are more than two messages in progress. In state D, only one command is in progress at the disk handler. Such detailed information would be very difficult to obtain without the use of finite state machines.

FSM Graph, Write Block

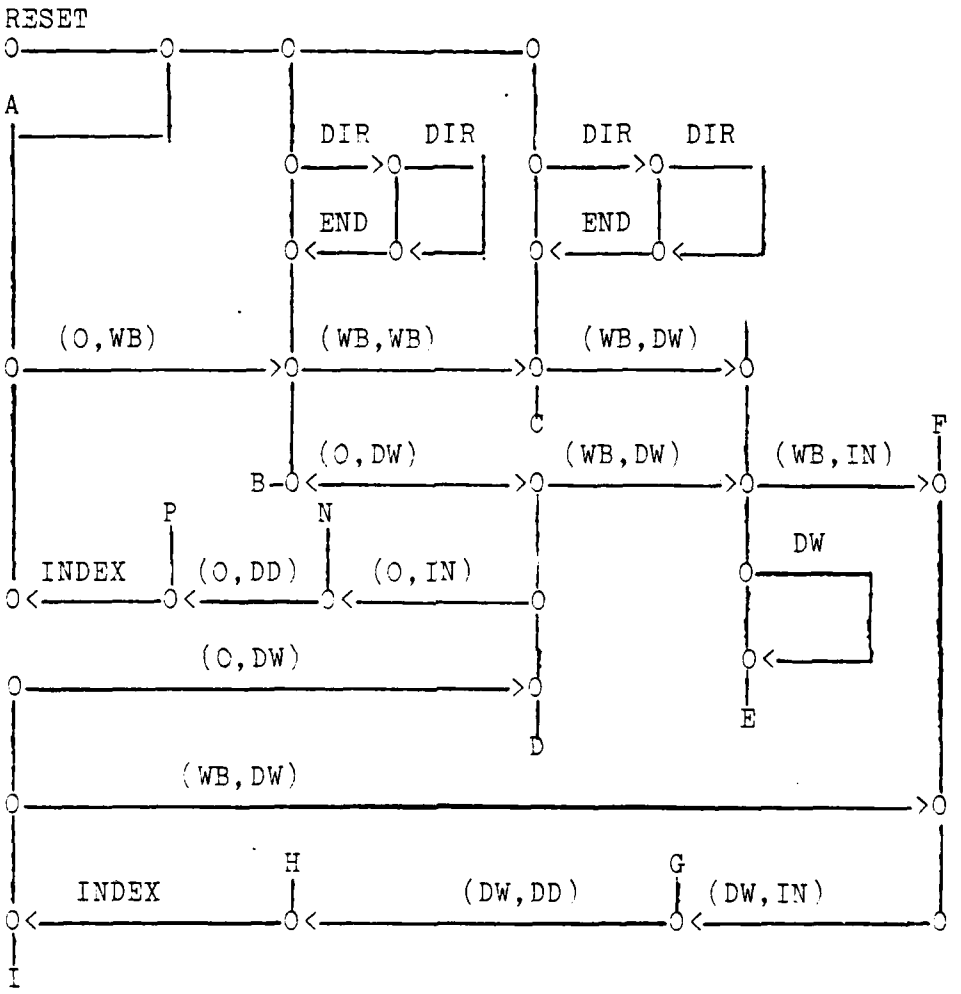


Figure 19: FSM Write Block, Two Messages in Progress

FSM Program, Write Block

```
FSM: WriteBlock
RESET:
  PREDICATE(Write[i+1]=Idle, Write[i]=Idle)
        CONNECT(A)
  PREDICATE(Write[i+1]=Idle, Write[i]=Writing)
        CONNECT(B)
  PREDICATE(Write[i+1]=Writing, Write[i]=Writing)
        CONNECT(C)
END RESET
A:    Write[i].(UserProcess->FileSystem, WriteBlock)
 B:   Write[i].(FileSystem->System, Directory)
  B1:  Write[i].(FileSystem->System, Directory)
       CONNECT(B1)
  B1:  Write[i].(FileSystem->System, EndDirectory)
       CONNECT(B)

 B:   Write[i+1].(UserProcess->FileSystem, WriteBlock)
   C: Write[i].(FileSystem->DiskHandler, DiskWrite)
      CONNECT(E)
   C:   Write[i].(FileSystem->System, Directory)
    C1: Write[i].(FileSystem->System, Directory)
        CONNECT(C1)
    C1: Write[i].(FileSystem->System, EndDirectory)
      CONNECT(C)

 B:   Write[i].(FileSystem->DiskHandler, DiskWrite)
   D: Write[i+1].(UserProcess->FileSystem, WriteBlock)
      CONNECT(E)

   D: Write[i].(System->DiskHandler, Interrupt)
      Write[i].(DiskHandler->FileSystem, DiskDone)
      Write[i].(FileSystem->FileSystem, WriteDone)
      CONNECT(A)
```

Figure 20: FSM Program, Write Block
(continued on the next page)

```
E:     Write[i].(System->DiskHandler, Interrupt)
  E1:  (User -> FileSystem, WriteBlock)
       CONNECT(E1)
F:     Write[i+1].(FileSystem->DiskHandler, DiskWrite)
 G:    Write[i].(DiskHandler->FileSystem, DiskDone)
 H:    Write[i].(FileSystem->System, WriteDone)

    J: PREDICATE(Write[i+1]=Writing, Write[i]=Busy)
       CONNECT(F)

    J: PREDICATE(Write[i+1]=Idle, Write[i]=Busy)
       CONNECT(D)

F:     Write[i].(DiskHandler->FileSystem, DiskDone)
 K:    Write[i].(FileSystem->System, WriteDone)
       INDEX(Write)

    M: PREDICATE(Write[i+1]=Writing, Write[i]=Writing)
       CONNECT(C)

    M: PREDICATE(Write[i+1]=Idle, Write[i]=Writing)
       CONNECT(B)

END-FSM
```

Figure 20: FSM Program, Write Block
(Continued)

4.3.4 Composite Model for Two Users

The same model [Figure 19] can be applied for performance analysis of two users. As was expected, the number of directory operations increased by 10%. This is because a smaller number of buffers is available for each user. Unexpectedly, however, the execution time of the transition (B->C) did not change. Further analysis revealed that in RIG there is no additional overhead associated with additional users. The file system makes no attempt to optimize disk input queues for the purpose of reducing the disk arm movement; therefore, the execution time in handling the request of users did not change. Likewise, the disk access time did not change (the transition E->F). Hence, another model is necessary to describe the file system in order to distinguish between requests of different users.

The same formalism that was used to describe pipelined computations is applied to multiple users. A typical sequence of events for two users is

```
WriteUser1[ i].(UserProcess1->FileSystem, WriteBlock)
WriteUser1[ i+1 ].(UserProcess1->FileSystem, WriteBlock)
WriteUser1[ i+2 ].(UserProcess1->FileSystem, WriteBlock)
WriteUser2[ i].(UserProcess1->FileSystem, WriteBlock)
WriteUser2[ i+1 ].(UserProcess1->FileSystem, WriteBlock).
```

Collected statistics indicate that transitions from state E to F occur three times as often as transitions from E to C occur. This suggests that the FileSystem receives user messages in bursts: first three messages from User1; second, three messages from User2. This explains that only every third message moves the disk arm from one user area to another user. (The number three is a default backlog in RIG, the number of messages to be queued in the receiver's input port. This explains why messages are received in bursts of three messages.)

FSM Graph, Write Block

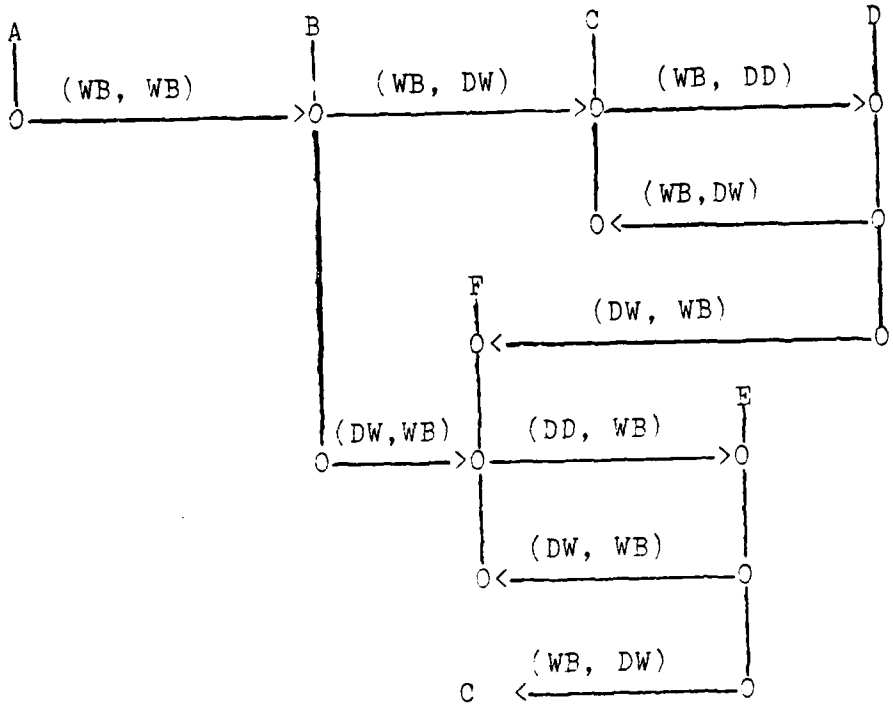


Figure 21: FSM Write Block, Two Users

4.3.5 Results

Throughout this section, various finite state machines have been constructed and applied to the same data to extract different kinds of information about the FileSystem in RIG. A model of a single user streaming data at maximum speed revealed various bottlenecks in the FileSystem. A simple model [Figure 18], without considering pipelining, pointed out the bottleneck- the handling of directories.

The third model [Figure 19] analyzed the relationship between pipelining of user messages and directory operations. The most interesting result was the model itself. Sometimes in the input queue of the FileSystem up to seven messages were queued by each user making it very difficult to model the system. A simple finite state machine modeling only two messages in progress was found sufficient for performance analysis of the file system.

4.4 Summary

Finite state machine models were found to be extremely valuable models for performance analysis of the RIG system. (The generality of the finite state machine formalism is discussed in Section 7.) Various examples were described and results for them were presented using the finite state machine formalism described in Section 3. The informal presentation was based on two examples from the RIG system: the virtual terminal and the file system. The first example presented a long sequence of messages produced by many processes communicating in full hand-shake. The second example presented pipelined computations that were described with a stream of messages containing many activities overlapped in time.

The first example was the initialization of a terminal in RIG. Four hundred messages were passed among fifteen processes. All computations were performed on a single processor. Although this was an extreme case that reduced computation of a potentially parallel program to a sequential program, this kind of computation is found in various initialization scenarios of other multi-process systems.

The initialization of a terminal in RIG produces a long sequence of message. To model this sequence, I introduced four levels of hierarchy: FSM(ReadBlock), FSM(ReadFile), FSM(CreateProcess) and FSM(InitializeTerminal). Statistics of the file system (FSM(ReadBlock) were expressed in terms of the global model (FSM(InitializeTerminal). This suggested an optimization of the high level protocol to keep process definition tables in memory, thereby avoiding file accesses entirely. This kind of optimization would be very

difficult to obtain without the use of finite state machines.

The second example was a user streaming data to disk. This produced a sequence of activities that were overlapped in time. The separate disk controller allowed the overlap of disk accesses with CPU computations. In describing the composite model of computations, I used the knowledge of the system to identify a small number of states characterizing the system.

To reduce the number of states in composite models, I successfully used three kinds of transitions: (1) FSM, characterizing a long sequence of messages; (2) PREDICATE, characterizing the exact system state as a vector of process states; (3)INDEX, characterizing a limited number of messages in a stream- a user view of computations in the system. The suitability of those transitions was tested by real measurements. Although many messages were outstanding, the model considered only two messages. This was sufficient to explain the overlap between the CPU and disk. The same model was also applied to analyze two users. Then, I described a different model in an attempt to analyze the multiplexing abilities of the file system.

5. Implementation

5.1 Introduction

The performance monitoring system ("the monitor") was implemented on a stand alone minicomputer (Xerox Alto) connected to the Ethernet, a 3 MHz broadcast network [Metcalfe 76]. RIG runs on two Data General Eclipse computers that are also connected to the Ethernet. The monitor receives from RIG time-stamped messages, process run times and swapping information. The performance evaluator describes finite state machines that identify events of interest in the execution trace of the system. The monitor calculates statistics of the abstract model and presents them to the performance evaluator [Figure 22].

The chapter has three major sections: Section 5.2 describes the interface to the performance evaluator. (An appendix contains a complete list of user commands.) Section 5.3 describes a technique for data gathering in RIG. This technique can be applied to other systems in which interprocess communication is well-defined. These systems are implemented in a style which is very close in spirit to either a message-based model or to a procedure-based model [Lauer et al., 79]. Section 5.4 outlines parsing of finite state machine descriptions into state-transition tables. The major effort here is to support the same symbols that are used both for programming of communicating processes and for describing finite state machines.

The purpose of this chapter is to demonstrate the feasibility of the performance monitoring system that is based on the use of finite state machines. This chapter describes the implementation of the monitor. In order to show that a finite state machine is an adequate model for the performance analysis of communicating processes, I have introduced a new formalism, tested it on a real system, and used the results to support the value of finite state machines. Chapter 3 introduced a finite state machine model of computation and described various time intervals that can be computed from such models for message-based systems. Chapter 4 applied the formalism to RIG and used the results to support the position that finite state machines are practical models for performance analysis.

5.2 User Interface

The monitor provides a command language for a user. By entering a command, the user affects the system state. The system then prompts the user for various parameters. Typically, a measurement experiment consists of two steps: data collection and data analysis. First, a user initiates

the data collection with the command ProduceTrace:

```
ProduceTrace
 =>host:
 =>time:
 =>file:
```

It requires three parameters: the host number (or the list of host numbers) of the system being measured; duration of the experiment in seconds; and name of the file that stores the trace data. Next, the user performs data analysis on the trace file with the command UseTrace:

```
UseTrace
=>trace-file:
```

A set of other commands helps the performance evaluator in data selection and analysis. For example, the command FSMLoad initiates data analysis with finite state machines.

```
FSMLoad
=>Trace (Yes or No) ?
=>InputFile:
=>OutputFile:
```

The command FSMLoad requires a name of the input file that contains symbolic description of finite state machines. The Trace option, when enabled, prints all state-transitions and their statistics. The OutputFile option directs finite state machine statistics to the specified file.

Several commands have been implemented to aid the user in describing finite state machines. The command SelectTriples produces a trace of chosen messages. Each message is defined with a triple consisting of a sender, receiver, and the message identifier.

```
SelectTriples
=>sender:
=>receiver:
=>message:
```

All options provide a consistent default value and help facility (for "help" the user types the "?" key, for default the user types the "return" key). Preparing the system state for an experiment may be a lengthy and tedious process. The use of a command file is a convenient way of automating the experiment. It contains the user's transcript as if he were interacting with the system. Since all the commands only prepare the monitor for data collection and analysis, a separate command is necessary to actually start the experiment. The command RUN then performs the experiment.

The same monitor has proven to be useful for other users

that are not interested in finite state machine models. Simple modifications of the disk handler allowed us to trace disk operations including both user file accesses and process page swapping.

5.3 Statistics Gathering

This section describes a technique for data gathering in RIG. The technique can be applied to other systems in which interprocess communications are well-defined. These well-defined interfaces allow the monitor to collect statistics selectively, at a very low cost. (In other systems, data collection can be very costly. For example, the General Trace Facility (GTF) consumes 30% of the CPU time in the system [IBM VS].)

To support the data collection, the system was modified in two places: the network handler and message-queueing. The modification to the network handler was made to provide a new type of service: to send system buffers over the net upon request; the modification to the message-queueing was made to store all recent messages in a circular buffer. The monitor then sends a request for statistics contained in the circular buffer.

The system's overhead in collecting statistics is small: only 3 milliseconds are required to send the circular buffer over the net, and 0.5 milliseconds to store a message in the buffer. The low cost in collecting statistics is due to two factors: acquiring statistics with a special "spying" protocol that is implemented within the interrupt level of the system, and a clear separation between a fixed message header and a message buffer.

The special protocol was made possible by placing the burden of reliable transmission on a dedicated computer executing the monitor. The monitor sends a request for statistics. After some time, if no reply has arrived, the monitor retransmits the request. The RIG system has only to pass a pointer to the link handler and start the output operation.

The clear separation between a fixed message header and a message buffer allows the RIG system to fetch the message header efficiently. A message header contains sender and receiver process numbers, the message identifier, two data words, and three time stamps. Consequently, for each message the RIG system performs a major data reduction of 512 bytes (a maximum message buffer size in RIG) at a low cost.

The size of the circular buffer is determined by the size of interval within which the monitor sends a new request of statistics and by the maximum number of messages being

queued by the system during that period. In RIG, I have chosen to collect statistics every second; in this case, the circular buffer of size 1K is sufficient since the system can queue at most 100 messages in one second.

The monitor, executing on a different computer on the Ethernet, periodically sends a request for statistics contained in the circular buffers and produces a textual trace of events that are ordered in time. In the case of communicating processes residing on different computers, the monitor sends a request for statistics to all systems that run processes of interest. To produce a time ordered trace of events, first, the monitor synchronizes clocks of communicating computers. (Although we can not achieve an exact synchronization of distributed clocks [Lamport 78], for the purpose of performance measurements, we approximate the error introduced by a 3 MHz local network.) Next, the monitor merges all events in the order of their appearance. Any two events that occurred at the same time (because of the finite resolution of the measurement clock) appear in arbitrary order.

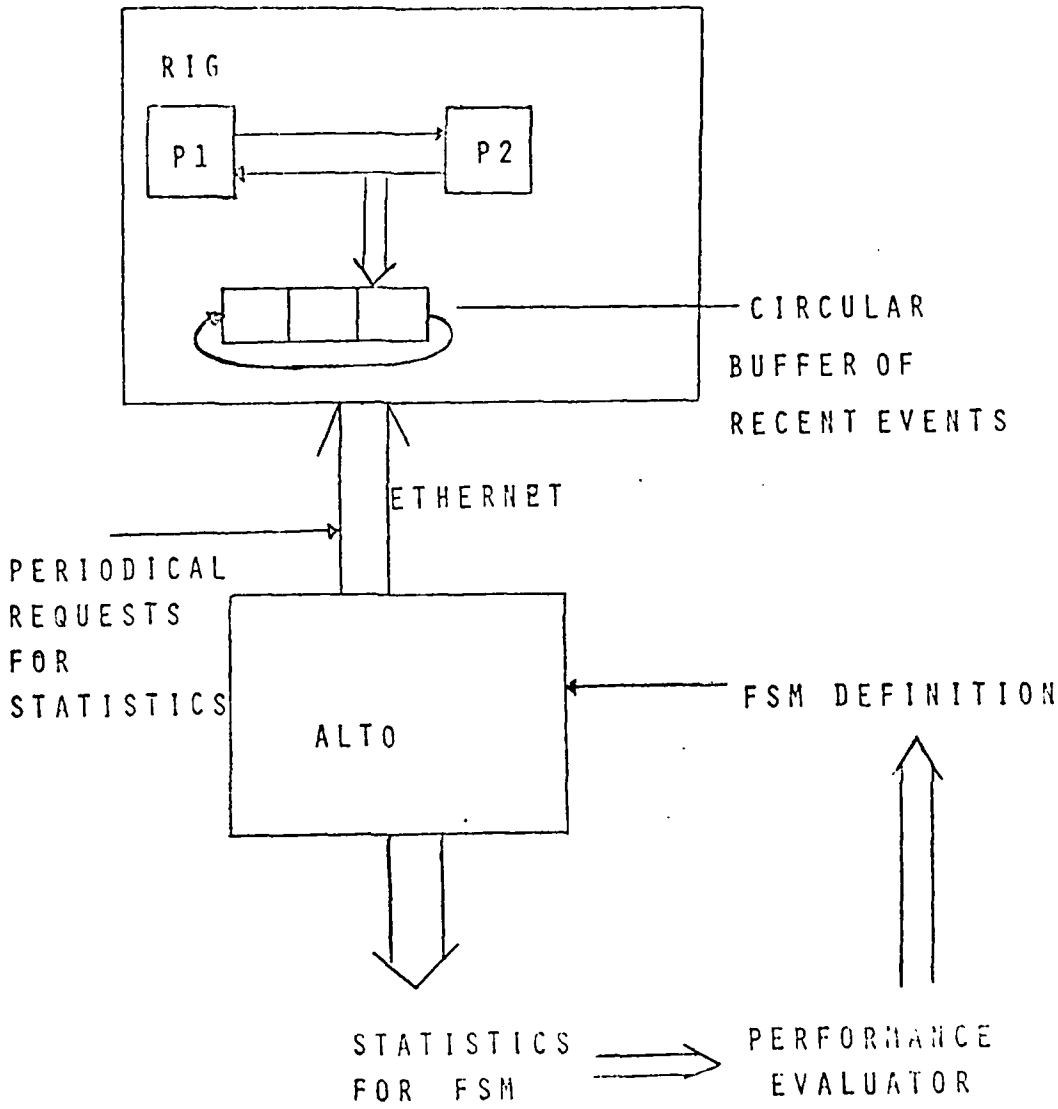


Figure 22: Implementation

5.4 Interpretation of Finite State Machines

This section outlines how finite state machine descriptions are parsed into state-transition tables. Here, the major effort is to support the same symbols that are used both for programming of communicating processes and for describing finite state machine models of computation.

The performance evaluator describes a finite state machine using symbolic references to RIG processes, messages and lower-level finite state machines. It is important to provide a uniform interface both for programming and monitoring. We should not expect the programmer to debug and to optimize the performance of his program through the use of memory dumps, loader maps, machine addresses and similar diagnostic tools [Batson 76]. To provide the uniform interface, the monitor uses the same standard header files used in the actual code of processes. In RIG the standard headers map some process names into numbers. In the case of processes not having fixed process numbers, the monitor requests the user intervention. Similarly, symbolic debuggers for multiprocess systems require the user specify the process number of the program [NSW MSG].

The internal representation of a finite state machine has a state transition table and a pointer to a higher-level finite state machine. Each transition has a message descriptor or a pointer to a lower-level finite state machine. When the transition occurs, statistics are calculated and stored. If a transition occurs in a lower-level finite state machine, the higher-level model accumulates statistics according to the rules that were defined for a sequence of events (Chapter 3).

5.5 Summary

This chapter demonstrated the feasibility of a monitor using finite state machines by describing a particular implementation. The main achievements of this implementation are simplicity and low cost in collecting statistics. The modifications of the existing system were very simple; what was required was to support a single request for reading system buffers and to store all recent events in a circular buffer. This type of implementation is possible for local networks that support high bandwidth data transmission.

In the case of a large number of computers connected to a local network, the monitor running on a single computer could become a bottleneck. For such networks, the monitor must be modified by separating the data collection and data filtering programs into a separate package. This package is

then placed on some computer in the network; thereby significantly reducing the amount of data flowing to the monitor. A similar approach has been used in the METRIC system [McDaniel 77].

The METRIC user views the world in three portions: a probe in the user's program, an account that collects information from the probe, and an analyst that processes the information and presents it in an intelligible format. Measurement events are those data that the probe transmits to the account, and which are subsequently processed by the analyst. The user's program and the probe live in a machine that is independent of the account and analyst's machine. This independence plays an important role in the robustness and efficiency of METRIC. Different from the monitor reported here, METRIC initiates probes (or events) at user selected places in a program. Consequently, METRIC has the ability to monitor a large number of computers, specifying a small number of events within each. Data analysis is performed by special purpose user programs. METRIC supports only a general purpose utility package to write data reduction programs.

A monitor reported here is a higher-level system than METRIC. A user is provided with a command language to initiate various experiments. Some commands are used to collect statistics, other commands are used to analyze the data. One of the commands is to use finite state machine descriptions to select events of interest out of the execution trace of the system. The performance evaluator describes these finite state machines using the same symbols that were used in programming the system.

6. Other Uses of Finite State Machines

6.1 Introduction

This chapter describes how finite state machines has benefited two areas: (1) validation of reliable transmission protocols and (2) optimized implementation of high-level protocols. In the first area, finite state machines describe a situation where the network servers which implement the reliable transmission protocol in RIG enter a loop of states causing each packet to be retransmitted twice.

In the second area, finite state machines helped to identify two different parts of the code within communicating processes: the first part deals with flow of exceptional messages modeled by many state-transitions in finite state machines; the second part deals with the common flow of messages modeled by fewer state-transitions. This observation suggests an optimization to support the most common case of the message flow. Instead of sending a message, a process may choose to perform computations locally.

The purpose of this chapter is to further motivate the reader in using finite state machine models of computation for the design, implementation and performance analysis of communicating processes. Chapter 3 introduced the formalism for descibing finite state machines for the analysis of communicating processes. Chapter 4 demonstrated the value of finite state machines by describing results obtained for the RIG system.

6.2 Reliable Communications Protocol

6.2.1 Introduction

This section uses the finite state machine formalism to describe a situation where the network servers which implement the reliable transmission protocol in RIG [Feldman et al., 78] enter a loop of states causing each packet to be retransmitted twice. The problem has been discussed but never described formally [Rovner 78]. Although the composite model having the loop of states is complex, it provides a language for the user to define conditions (or transitions in the composite model) that cause those retransmissions.

This section has three major subsections: Subsection 6.2.2 describes two elementary finite state machines: one modeling the sender and another the receiver. Subsection

6.2.3 describes the composite finite state machine modeling two senders and two receivers. Subsection 6.2.4 formally defines the system states that cause entrance into the loop of retransmission and those that cause exit from the loop.

6.2.2 Elementary Models of Sender and Receiver

Two elementary finite state machines are described: one modeling the sender and another the receiver. The example is a sender streaming messages to a receiver. For each packet having a correct sequence number, the receiving network server sends back an acknowledgment. For each received acknowledgment, the sending network server flushes the buffer holding the outstanding message.

The sender starts in "idle" state Idle [Figure 23], and upon receiving a message from UserProcess routes it over the net, starts a timer and moves to "waiting for an acknowledgment" state Wait. In the decision state Wait, either the timer expires and the sender moves to "message has timed out" state Timeout, or the acknowledgment arrives and the sender moves to "message acknowledged" state Ack. To retransmit the message in state Timeout, the sender sends the message once again, starts the timer and moves back to state Wait; to complete the protocol, in state Ack the sender posts the buffer and moves back to state Last.

The model of the receiver is more complicated than one of the sender. A message may arrive out of order and be rejected. Those decisions are based upon the status of the finite state machine modeling the message with a lower sequence number. The composite model contains transitions that depend on the state of the lower-level finite state machine. To describe those transitions, I use the construct PREDICATE (see Section 3.4.3). For example, the transition

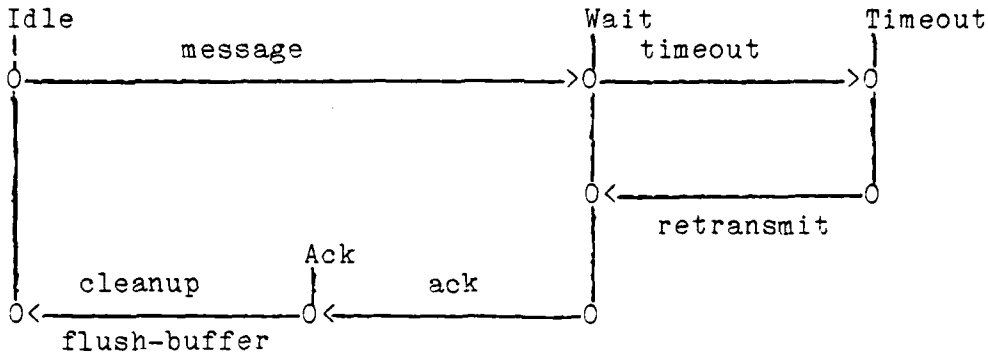
PREDICATE(Receiver[i-1] > MSGTran)

occurs when Receiver[i-1] is in the state that follows MSGTran (e.g. Accepted or Last).

Receiver[i] starts in state Idle. If the receiver model has accepted the previous message (the model Receiver[i-1] is either in state Accepted or in Last), Receiver[i] moves to "available to receive a message" state Available. Another alternative in state Idle is to receive the next message in transit which moves Receive[i] to "temporary in transit" state TempTransit. If the previous message has been received correctly (PREDICATE([i-1]>MSGTran), Receiver[i] moves to state MSGTran; otherwise, an error occurs since the message has arrived out of sequence. The message is rejected and Receiver[i] moves to state Reject. In state Available, having accepted a message, Receiver[i] moves to

state Accepted where it sends back an acknowledgment and enters the last state.

FSM Graph, Sender



FSM Graph, Receiver

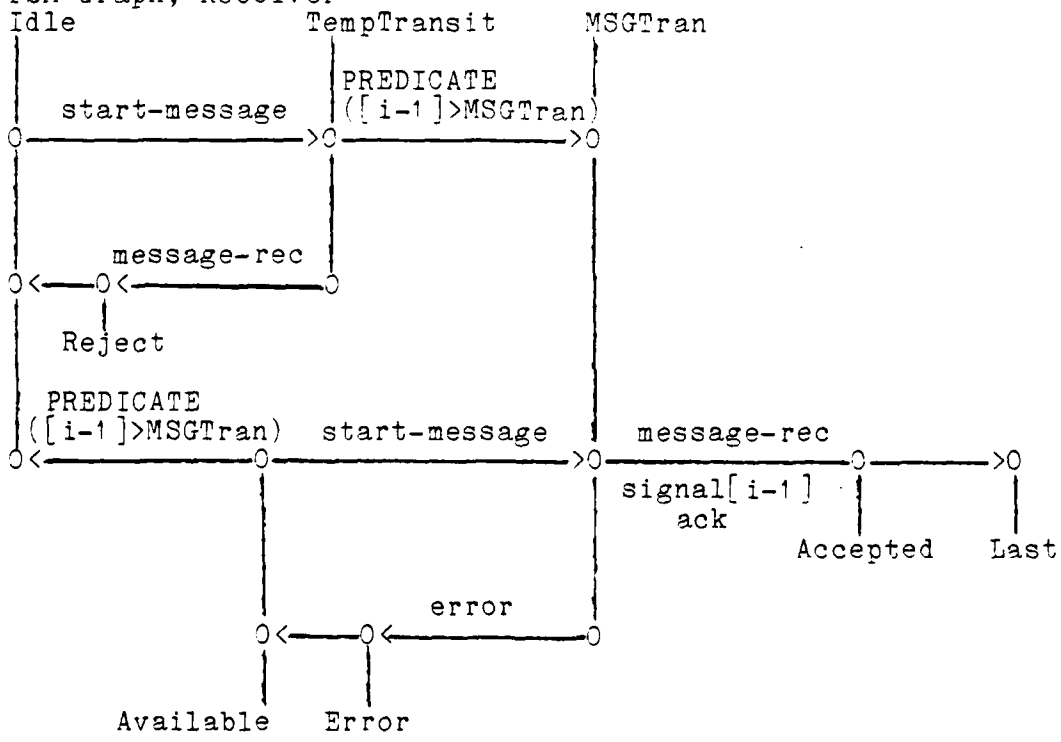


Figure 23: FSM Communications Protocol, Sender and Receiver

Ideally, the following sequence of events occurs for two messages in transit:

```
                Sender[ i].message
Available:      Receiver[ i].start-message
                Sender[ i+1 ].message
Idle:           Receiver[ i+1 ].start-message
MSGTran:        Receiver[ i].message-rec
TempTransit:    Receiver[ i+1 ].signal
                Sender[ i].ack
                Sender[ i+2 ].message
Idle:           Receiver[ i+2 ].start-message
MSGTran:        Receiver[ i+1 ].message-rec
TempTransit:    Receiver[ i+2 ].signal
                Sender[ i+1 ].ack
```

According to the specifications, the following sequence of events, having retransmissions for every message, might occur:

```
                        Sender[ i].message
(1)  Available:         Receiver[ i].start-message
(2)  MSGTran:           Receiver[ i].error
                        Sender[ i+1 ].message
     Idle:              Receiver[ i+1 ].start-message
(3)  TempTransit:       Receiver[ i+1 ].msg-receive
(4)                     Sender[ i].timeout
     Available:         Receiver[ i].start-message
     MSGTran:           Receiver[ i].message-receive
                        Receiver[ i+1 ].signal
                        Sender[ i].ack
(5)                     Sender[ i+2 ].message
     Idle:              Receiver[ i+2 ].start-message
     TempTransit:       Receiver[ i+2 ].msg-receive
```

The first retransmission (the transition on line 4) occurs as a result of an error in the transmission media (on line 2). Consequently, the message [i+1], being out of sequence (line 3) is rejected. Later, the message [i] is successfully received and acknowledged (the sequence of transitions starting on (line 4). Now, the newly arrived message [i+2] (line 5) is posted but it will eventually be rejected because the message [i+1] has not been received. This may occur in a loop causing every message to be retransmitted twice.

Note that by extending the receiving window to accept up to K messages out of sequence, the problem may still occur when the message [i+k] has been posted before the message [i] is retransmitted. One solution is to withhold sending the message [i+2] until the message [i+1] is delivered successfully. Another solution is to retransmit all messages in transit with sequence number "j" such that "j" is greater than "i" for all messages [i] that were lost.

6.2.3 Composite Model for Sender and Receiver

Although the composite model is a complex program, the transitions in the composite model help to discern when the problem is happening (not every message loss causes the sequence of events) and why the problem sometimes disappears as a result of a different system load.

I describe a composite model for two messages in progress using four finite state machines: two describing the sender and two the receiver. To simplify the composite model, I include only those system states that are of significant duration. For model of the sender, the following two combinations are of significant duration:

(Sender[i+1]=Idle, Sender[i]=Waiting)
(Sender[i+1]=Waiting, Sender[i]=Waiting)

All other combinations are immediately followed by internal actions of the network server. The receiver model has four combinations of significant duration:

(Receiver[i+1]=Idle, Receiver[i]=Available)
(Receiver[i+1]=Idle, Receiver[i]=MSGTran)
(Receiver[i+1]=TempTransit, Receiver[i]=Available)
(Receiver[i+1]=TempTransit, Receiver[i]=MSGTran).

Overall, there are eight possible combinations of system states; out of them, two combinations are illegal:

(Sender[i+1]=Idle, Sender[i]=Waiting,
Receiver[i+1]=TempTransit, Receiver[i]=Available)

(Sender[i+1]=Idle, Sender[i]=Waiting,
Receiver[i+1]=TempTransit, Receiver[i]=MSGTran)

Consequently, the composite system model has only 6 states: A, B, C, D, E, and F [Figure 24].

```
FSM: Sender-Receiver

A:
Predicate(Sender[ i+1 ]=Idle, Sender[ i]=Wait,
          Receiver[ i+1 ]=Idle, Receiver[ i ]=MSGTran)

 A1:     // message [ i]  was received

          (Receiver[ i ]=Accept)
          (Receiver[ i ]=Last)
          (Sender[ i ]=Ack)
          (Sender[ i ]=Last)

          INDEX(Sender, Receiver)
          CONNECT(RESET)

 A2:     // message [ i] was lost

          (Receive[ i ]=Error)
          (Receive[ i ]=Available)
          CONNECT(B)

 A3:      // message [ i+1 ] is posted

          (Sender[ i ]=Wait)
          (Receiver[ i ]=TempTransit)
          CONNECT(D)
B:
 Predicate(Sender[ i+1 ]=Idle, Sender[ i ]=Wait,
          Receiver[ i+1 ]=Idle, Receiver[ i ]=Available)

 B1:      // message [ i] timeout

          (Sender[ i ]=Timeout)
          (Sender[ i ]=Wait)
          (Receiver[ i ]=MSGTran)
          CONNECT(A)

 B2:      // message [ i+1 ] is posted

          (Sender[ i ]=Wait)
          (Receiver[ i ]=TempTransit)
          CONNECT(D)
C:        // message [ i] timeout
Predicate(Sender[ i+1 ]=Wait, Sender[ i ]=Wait,
          Receiver[ i+1 ]=Idle, Receiver[ i ]=Available)

          (Sender[ i ]=Timeout)
          (Sender[ i ]=Wait)
          (Receiver[ i ]=MSGTran)
          CONNECT(D)
```

Figure 24: Composite FSM , Sender-Receiver
(continued on the next page)

```
D:
Predicate(Sender[i+1]=Wait, Sender[i]=Wait,
          Receiver[i+1]=Idle, Receiver[i]=MSGTran)

 D1:      // first receive ack [i]

          (Receive[i]=Accept)
          (Receive[i]=Last)
          (Receive[i+1]=Available)
          (Sender[i]=Ack)
          (Sender[i]=Last)
          INDEX(Sender, Receiver)
          CONNECT(RESET)

 D2:      // first timeout message [i+1]

          (Sender[i+1]=Timeout)
          (Sender[i+1]=Wait)
          (Receiver[i+1]=MSGTran)
          CONNECT(E)
E:
Predicate(Sender[i+1]=Wait, Sender[i]=Wait,
          Receiver[i+1]=TempTransit, Receiver[i]=MSGTran)

 E1:      // message [i]  received and acknowledged

          (Receive[i]=Accept)
          (Receive[i]=Last)
          (Receive[i+1]=Available)
          (Sender[i]=Ack)
          (Sender[i]=Last)
          INDEX(Sender, Receiver)
          CONNECT(RESET)

 E2:
          // message [i] was lost

          (Receive[ i]=Error)
          (Receive[i]=Available)
          CONNECT(F)
F:
Predicate(Sender[i+1]=Wait, Sender[i]=Wait,
          Receiver[i+1]=TempTransit, Receiver[i]=Available)

         // reject message [i+1] out of sequence

          (Receive[i+1]=Reject)
          (Receive[i+1]=Idle)
          CONNECT(C)
```

Figure 24: Composite FSM, Sender-Receiver
(continued)

```
RESET:

Predicate(Sender[ i+1 ]=Idle, Sender[ i ]=Wait,
          Receiver[ i+1 ]=Idle, Receiver[ i ]=MSGTran)
   CONNECT(A)
Predicate(Sender[ i+1 ]=Idle, Sender[ i ]=Wait,
          Receiver[ i+1 ]=Idle, Receiver[ i ]=Available)
  CONNECT(B)

Predicate(Sender[ i+1 ]=Wait, Sender[ i ]=Wait,
          Receiver[ i+1 ]=Idle, Receiver[ i ]=Available)
  CONNECT(C)

Predicate(Sender[ i+1 ]=Wait, Sender[ i ]=Wait,
          Receiver[ i+1 ]=Idle, Receiver[ i ]=MSGTran)
  CONNECT(D)

Predicate(Sender[ i+1 ]=Wait, Sender[ i ]=Wait,
          Receiver[ i+1 ]=TempTransit, Receiver[ i ]=MSGTran)
  CONNECT(E)

Predicate(Sender[ i+1 ]=Wait, Sender[ i ]=Wait,
          Receiver[ i+1 ]=TempTransit, Receiver[ i ]=Available)
  CONNECT(F)
```

Figure 24: Composite FSM, Sender-Receiver
(continued)

6.2.4 Results

To describe the entire model would be like trying to describe a program in English; instead, I describe only those system states which cause retransmissions.

In state F, message [i+1] arrives out of order, therefore, it is rejected. The following conditions cause the entrance of the state F. In state E there are two messages in transit: [i] and [i+1]. If message [i] is lost, the model enters "message [i] is lost, and message [i+1] is still in transit" state F. Clearly, message [i+1] will arrive at Receiver[i+1] out of order and will be rejected. This is the type of a situation of prime interest in this section. The following sequence of state-transitions enters state F:

E -> E2 -> F

In state F, message [i+1] is rejected moving the model to "both messages are lost" state C. Then, Sender[i] times out message [i] moving to "message [i] is in transit" state D. In state D there are two alternate transitions: to state D1 and to D2. The state D1 continues moving the model through the loop of states causing retransmissions. In state D1, message [i] arrives correctly at Receiver[i] which acknowledges Sender[i], thereby completing the protocol for message [i]. In addition, Receiver[i+1] moves to state Available. In this case, the INDEX operation is applied both for the sender and receiver. In state RESET, depending on the status of message [i+1] (which was message [i+2] prior to the INDEX operation), there are three alternate transitions: to state B, C, or F. In all these cases, the message [i] is lost. If the message [i+1] is already in transit, the model again enters the problematic state F. In summary, the following state transitions occur in a loop causing retransmission of each message:

F -> C -> D -> D1 -> RESET -> F

There are two transitions escaping from the loop: in state D and RESET. In state D, the transition to state D2 (retransmission of message [i+1]) moves the model back to the correct state E. In state RESET, the transition to state C continues moving the model through the loop; the transition to state B has two alternatives: transition to state B1 guarantees the escape from the loop, and transition to B2 moves the model back to state D.

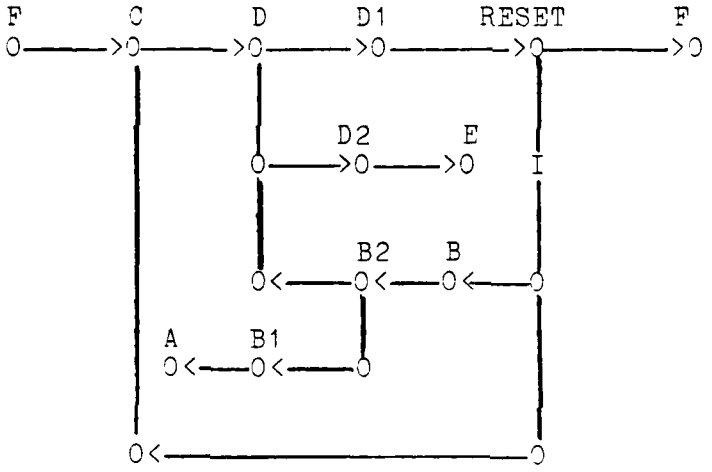
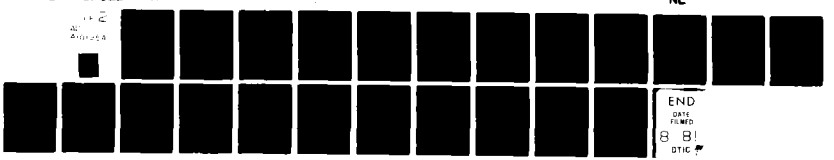


Figure 25: FSM for the Retransmission Loop

AD-A101 954 ROCHESTER UNIV NY DEPT OF COMPUTER SCIENCE F/G 9/2
PERFORMANCE EVALUATION OF COMMUNICATING PROCESSES.(U)
MAY 80 I GERTNER N00014-78-C-0164
UNCLASSIFIED TR-76 NL



In conclusion, there are three loops of states causing rejection of messages. The first loop rejects messages in state F and later retransmits them in state C; the second loop retransmits in state B2; the third loop retransmits in state C.

1) F -> C -> D -> D1 -> RESET -> F
2) D -> D1 -> RESET -> B -> B2 -> D
3) C -> D -> D1 -> RESET -> C

Note that two escape conditions through states D2 and B1 were previously described as a way to avoid retransmissions. The transition (D->D2) is enforced by always retransmitting messages with higher sequence number; likewise, the transition (B->B1) is enforced by do not sending messages with higher sequence number until an acknowledgment has been received.

6.3 Optimization of High-Level Protocols

6.3.1 Motivation

One advantage in multi-process systems is the flexibility offered by easy modification of processes without reassembling the whole system [Parnas 72]. To achieve a flexible system, we strived to hide implementation decisions within each process in RIG. Unfortunately, such a clean decomposition of a system into processes increases the number of messages. This section proposes an optimization that significantly reduces the number of messages in the system. Instead of sending a message, a process may choose to perform computations locally. An analogy is an optimizing compiler, which, in an attempt to save the procedure call overhead, inserts the procedure body into the code of the calling program; or, in some other cases, a highly specialized and efficient control transfer.

Finite state machine models of computation helped to identify two different parts of a process code: the first part deals with flow of exceptional messages modeled by great many of state-transitions in finite state machines; the second part deals with the common flow of messages modeled by a small number of state-transitions. This observation suggests an optimization that is appropriate for the most common case of the message flow. The optimized implementation requires fewer processes and messages to support the same computation; consequently, the system's overhead and the working set size are significantly reduced.

The purpose of this section is to motivate the reader further in using finite state machine models of computation for the design of communicating processes. There is always a tension between monolithic and modular structure of systems. As the number of processes in the system increases, the overhead in interprocess communication also increases. This section proposes a method to reduce the system overhead by centralizing the code and state of every process dealing with the common message flow.

6.3.2 An Optimization of the PDP-10 Telnet Protocol

In RIG, the PDP-10 Telnet facility requires five processes: Telnet, TenServer, TTYInput, TTYOutput and DCU; the virtual terminal requires five processes: TerminalInput, LineHandler, Pad, TerminalOutput and DCU [Lantz 80]. Overall nine processes support the flow of messages between the PDP-10 and user. To simplify the example, we consider a user program running on the PDP-10 and printing data on the RIG terminal. In this particular case, one process that has obtained the state and code from four other processes is

sufficient to support the protocol. The following information is provided by the four processes:

1) TTYInput provides the code and state of the line that handles incoming characters from the PDP-10.

2) TenServer provides the code and tables for multiplexing different users sharing the same physical line.

3) Pad provides the code and data structures for displaying one line on the terminal.

4) TerminalOutput provides code and state for handling the terminal screen.

In RIG, the code for handling the common case of the message flow is very simple; most of the code deals with handling of exceptional conditions. For example, TTYInput is concerned with errors occurring on the physical line. TenServer handles the flow of incoming characters. In addition, it recognizes various control characters having a special meaning on the PDP-10. Pad maintains the mapping between virtual and physical screens. Recognizing those conditions is easier than handling them. For example, if a user types a special character "<CTL>S", the system's activities drastically change: TenServer simulates the meaning of the special character by disabling output on the virtual terminal in RIG. In addition, it sends the character to the PDP-10 and waits for special user actions (<CTL>Q) to resume output of the user program. In this case, recognition of the special character was easy but the handling was more difficult requiring establishment of the new state. Another example requiring complex actions of the system is a user changing the configuration of the RIG terminal causing the Pad process to cease the display of data on the terminal screen.

In summary, the optimization of a high-level protocol is possible for the common message flow at the expense of the exceptional flow. Here, the hypothetical example was the PDP-10 Telnet protocol (it was never implemented). The basic assumption was that most of the code establishes the initial state of processes and handles exceptional conditions but only a very small portion of code supports the common flow of messages. To optimize the flow, I suggested centralizing the state and code dealing with the common case. The functions that previously required a few processes are then performed by a single process. To apply those ideas to other systems might be very difficult. Nevertheless, for systems that are designed using finite state machine models of computation, we can again identify the initialization of a finite state machine and gain in efficiency of the implementation.

6.4 Summary

In addition to performance analysis, finite state machines have benefited two areas: (1) validation of reliable transmission protocols and (2) optimized implementation of high-level protocols.

In the first area, I used finite state machines to describe a situation where the network servers, which implement the reliable transmission protocol in RIG, enter a loop of states causing each packet to be retransmitted twice. The composite model contained the loop of state transitions causing retransmission of each message and the alternate state-transitions causing exit from the loop. To reduce the number of states in the composite model, I described only those states that are of significant duration in execution of the system. This is a novel idea in describing composite models. The suitability of two transitions, INDEX and PREDICATE, was tested in new applications.

In the second area, finite state machines helped to identify a large portion of the code that deals with initialization and handling of exceptional messages but only a small portion of the code deals with the common case of message flow. An optimization was described for a simple case, where a process, instead of sending a message, may choose to perform computations locally, thereby significantly reducing the number of messages. These two examples further support the value of finite state machines for designing, implementing and analyzing the performance of communicating processes.

7. Conclusions

7.1 Results

The thesis presents a new method for performance analysis of communicating processes based upon a finite state machine model. An experimental performance monitoring system was implemented and applied to the analysis of RIG, a message-based distributed operating system.

First, I described the basic properties of messages traces by introducing a small number of time stamps (Birth, Start and Finish) that were used to calculate intervals of interest (Execution, Interval and Delay). Analogous intervals were also defined for a sequence of messages.

Further, I introduced a finite state machine model describing the semantics of the message traces. The time intervals that were used to describe messages were extended to describe events defined by state-transitions in the finite state machine model. Elementary finite - state machines described a single process representing a sequential program; composite finite state machines described a group of processes representing a parallel program. To reduce the number of states in composite models, I introduced three new kinds of transitions: (1) FSM describes a long sequence of messages, (2) PREDICATE describes the exact system state as a vector of process states, and (3) INDEX describes a limited number of messages in a stream. These transitions were used to describe a composite model of the system.

The quality of RIG has improved substantially due to the use of the monitor. In this dissertation, I described two kinds of examples: The first example dealt with long sequences of messages produced by many processes communicating in a full hand-shake style. The second example dealt with pipelined computations that were described with a stream of messages.

To describe a long sequence of messages, I used a four levels hierarchy of finite state machines. This allowed the performance evaluator to concentrate on very detailed computations while retaining the overall statistics of the system. This kind of information would be very difficult to obtain without the use of finite state machines.

The second example was concerned with pipelined computations. First, I described a separate finite state machine for each message in progress. The composite model then described system states as a vector of states of finite state machines each modeling one message in progress. To reduce the number of states in the composite model, I used the transitions PREDICATE and INDEX to describe a limited number of messages in a stream. These transitions were used

to describe a small subset of system states (which represent a user view of computations in the system). This is a novel idea in describing composite models and is based on the observation that only a small number of system states are actually reached during the system's execution.

Composite models expressing the user view of computation were applied to validate reliable transmission protocols. I described a situation where the network servers which implement the reliable transmission protocol in RIG enter a loop of states causing each packet to be retransmitted twice. Again, both transitions INDEX and PREDICATE were successfully applied in describing transitions between the chosen set of system states. The problem of retransmission was then formally described in terms of system state-transitions.

Finite state machine models were found to be extremely valuable and practical models for the performance analysis of communicating processes. Although a group of processes constitute a parallel program that is characterized only by a partial ordering of events, in our experience with RIG only a small fraction of that partial ordering is exercised. Under changing load conditions, the system passes through a large number of states and quickly stabilizes to a new set of states characterizing the system under each new load. The surprising result was that the total number of states describing the average behavior of the system remained small for a wide range of the load. This observation directs the performance evaluator in a search of a small set of states that occur often in the execution of the system and are of significant duration.

The growing interest in message-based computing and in formal description of communication protocols suggests that many future systems will be implemented or at least designed using finite state machines. In those cases, the performance evaluator will immediately have accurate finite state machine models available for the performance analysis. For other systems, describing finite state machines may be very difficult or even impossible (due to the large number of states). Although systems are implemented as a collection of parallel programs, a well-designed system is characterized by sequential behavior at a higher-level of abstraction. Hence, one should be able to apply finite state machines to describe sequences of events.

7.2 Disadvantages

Performance monitoring with finite state machine models of computation has two drawbacks: (1) limited scope of applications and (2) difficulty in programming those models. For example, abstract models do not help in the search for computational bottleneck at the procedure level. I have witnessed situations where rewriting a single procedure in an assembler improved the overall system performance by 10%. In this case, the main problem was to find that procedure which was the bottleneck of the system. An abstract model describing the user view of computations in the system would frequently fail to include the bottleneck in the model.

Inventing concise models of complicated systems is an art. Many experiments, as well as deep understanding of the system, are required to debug the model of a computation. One difficulty is in finding a small subset of system states that occur often in execution of the system and are of significant duration. Another difficulty is in encoding the model into a machine-readable form. Although the language developed in the dissertation helps, other methods (which are beyond the scope of this work) need to be tried, e.g. a graphical drawing of a finite state machine and a compiler accepting this drawing would be potential assets to the performance evaluator.

7.3 Understanding Concurrency

Understanding concurrency is a topic of great interest for the theoretical computer science community [Fisher et al., 79]. Herein, I compare the use of parallel structures in RIG with other systems. In our experience with RIG, we have developed a set of guidelines (unenforced) which made the implementation and debugging of the system easier.

Several modern languages provide facilities which express parallelism in programs ([Lampson et al., 1980] and [Brinch-Hansen 75]). In RIG, the parallelism is static: processes are created or killed rarely, and this is done only at system initialization time. Consequently, a RIG process is a sequential program; the parallelism is achieved by having multiple processes. Other systems composed of processes representing sequential programs allow interprocess communication via shared variables [Peterson 79]. In RIG, processes share no data and communicate only via messages.

There are basically two styles of message communication: full hand-shake and message streaming. The message streaming is the only means of parallel computation. The purpose of those constraints is to further reduce the number of states in the system. Debugging this kind of computation

is easy: the flow control mechanisms guarantee a limited number of messages outstanding for every process; the full-hand shake limits the number of processes ready for execution. Analogous constraints in programming of parallel systems have also been used by other authors ([Mattheyses et al., 79] and [Yoeli 78]).

One can argue that a system composed of processes communicating in full hand-shake or in message streaming has less parallelism than would be possible by using a more liberal style of message-passing. It may be, however, that the system can not be implemented and debugged by any other style of communication, or it may not be practical with the available collection of tools and concepts (e.g. better debuggers, suitable for parallel processes).

7.4 Future Work

Accurate models of behavior are still by far the weakest link in all attempts to evaluate the performance of complex computer systems. In the case of multi-process systems in which processes communicate only via messages, system designers and implementers have much better intuitions on the behavior of the system. The designers use (or at least they should use) finite state machines to validate the correctness of those systems. In this dissertation, I applied finite state machines to analyze the performance of such systems. The use of crude finite state machine models having detailed specifications at the points of interest appears promising.

Clearly, progress in this area depends on the extent to which finite state machines can describe existing operating systems or can be applied to describe future systems. Many future systems will be designed and implemented with various automated tools using formal models of computation. (Today, the notable example is SARA, a simulation system using UCLA graphs and a very high-level language for design and analysis of new systems [Estrin et al., 78]). The same formal models can then be applied to performance analysis of those systems. We have done much of this for the RIG system using finite state machine models of computation. Our experience with a real system indicates that our methods are sound, that even a crude finite state machine model is adequate for finding performance bugs. The next step is to apply those ideas to other systems and to solve a wider range of performance problems.

Bibliography

[Anderson 1976] Anderson J. and Browne J. "Graph Models of Computer Systems: Application to Performance Evaluation of an Operating System", in Proc. International Symposium on Computer Performance Modeling, Measurement and Evaluation, Harvard University, 1976.

[Baker 78] Baker H. "Actor System for Real-Time Computation", Ph.D Thesis, Laboratory for Computer Science, MIT, 1978.

[Ball 1976] Ball E., Feldman J., Low J., Rashid R., Rovner P. "RIG, Rochester's Intelligent Gateway: System Overview", IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, December 1976.

[Ball 1979] Ball.E, Burke E., Gertner I., Lantz K., Rashid R., "Perspective on Message-based Distributed Computing", IEEE Proc. Computer Networking Symposium, Maryland, December, 1979.

[Bartlett 69] Bartlett K., Scantlebury R., Wilkinson P. "A Note on Reliable Full Duplex Transmission over 1/2 Duplex Links", Comm. ACM, Vol. 12, No. 5, 1969.

[Baskett 77] Basket F., Howard J., and Montague J., "Task Communication in Demos", ACM Proc. of the 6th Symposium on Operating Systems Principles, November 1977.

[Batson 76] Batson A. "Program Behavior at the Symbolic Level", Computer, November 1976.

[Bochman 1977] Bochman G. and Gescei J., "A Unified Method for the Specification and Verification of Protocols", Information Processing 77, Proceedings of the IFIP Congress 77(Toronto), 1977.

[Bochman 1977] Bochman G.and Jochim T., "Development and Structure of X.25 Implementation, Publication 292, University of Montreal, July 1978.

[Bochman 1978] Bochman G., "Finite State Description of Communication Protocols", in Proc. Computer Network Protocols Conf. Liege, 1978.

[Brinch-Hansen 1978] Brinch-Hansen "The Programming Language Concurrent PASCAL, IEEE Transactions on Software Engineering, SE-1, 1975.

[Campos 78] Campos I. and Estrin G. "SARA Aided Design of Software for Concurrent Systems", National Computer Conference, 1979, pp. 325-336.

[Chany 78] K.Chany and J.Misra "Specification, Synthesis,

Verification and Performance Analysis of Distributed Programs", TR-86, University of Austin, November, 1978.

[Cheriton 79] Cheriton D., Malcolm M., Melen L., and Sager G. "Thoth, a Portable Real-Time Operating System", Comm. CACM, February 1979.

[Dijkstra 1968] Dijkstra C. "Co-operating Sequential Processes", in Programming Languages, ed Genuys F., Academic Press, N.Y., pp. 43-112, 1968.

[Estrin 67] Estrin G. and Martin D. "Experiments on Models of Computations and Systems", IEEE Tran. on Electronic Computers, Vol. EC-16, No. 1, 1967.

[Estrin 78] Estrin G., "A Methodology for Design of Digital Systems- Supported by SARA at the age of one", AFIPS Proceedings, Vol. 47, 1978, pp. 313-324.

[Farber 73] Farber D. et al., "The Distributed Computing System", IEEE Proc. Compcon, 1978.

[Feldman 1971] Feldman J. and Sproul B., "System Support for the Stanford Hand-Eye System", Second International Joint Conference on Artificial Intelligence, London, September, 1971.

[Feldman 1977] Feldman J. "Synchronizing Distant Cooperating Processes", TR-26, Department of Computer Science, the University of Rochester, 1977.

[Feldman 1978] Feldman J., Low J. and Rovner P., "Programming Distributed Systems", Proc. of the National Conference of the ACM, Washington, DC, December 1978.

[Feldman 1979] Feldman J., "High Level Programming For Distributed Computing", Comm. ACM, July 1979.

[Gertner 79a] Gertner I. "Performance Evaluation of Communicating Processes", Proc. ACM. Conference on Simulation, Modeling and Measurement of Computer Systems, Boulder, Colorado, August 1979.

[Gertner 79b] Gertner I. "Performance Analysis of RIG", internal memo, Computer Science Department, University of Rochester, August, 1979.

[Gertner 79c] Gertner I. "RIG System Kernel", internal document, Computer Science Department, University of Rochester, October, 1979.

[IBM 76] IBM, "Systems Network Architecture, Format and Protocol Reference Manual: Architecture Logic", SC 30-312-0, IBM Corporation, White Plans, NY, 1976.

[IBM VS] IBM, OS/VS2 System Programming Library: Service Aids. Pub. No. GC28-06774, IBM Corp., White Plains, N.Y.

[Hansen 1978] Hansen P. "Distributed Processes: A Concurrent Programming Concept", Comm. ACM, Vol. 21, No. 11, 1978.

[Howard 1972] Howard J. and Alexander W. "Analyzing Sequences of Operations Performed by Programs", Program Test Methods, Prentice-Hall, 1973, p. 239-254.

[Lamport 78] Lamport L., "Time, Clocks, and the Ordering of Events in a Distributed System", Comm. ACM, Vol. 21, No. 7, July 1978.

[Lampson 80] Lampson B. and Redell D., "Experience with Processes and Monitors in Mesa", Comm. ACM, Vol. 23, No.2, February 1980.

[Lantz 1979] Lantz K. and Rashid R. "Virtual Terminal System", CR-46, Department of Computer Science, the University of Rochester, 1979.

[Lantz 80] Lantz K., "Uniform Interfaces to Distributed Systems", Ph.D Thesis, Computer Science Department, University of Rochester, 1980.

[Lauer 1977] Lauer C. and Needham H. "On the Duality of Operating System Structures", Xerox Palo Alto Research Center, 1977.

[Lausen 75] Lausen A Large Semaphore Based Operating System", Comm. ACM, Vol. 18, No. 7, 1975.

[Lucas 71] Lucas H. "Performance Evaluation and Monitoring", ACM Computing Surveys, Vol. 3, No. 3, September, 1971.

[MacDougall 78] MacDougall M. "The Event Analysis Program", National Computer Conference, 1978.

[Mattheyses 79] Mattheyses R., Conry S. "Models for Specification and Analysis of Parallel Computing Systems", ACM. Proc. Conference on Simulation, Measurement and Modeling of Computer Systems", Boulder, Colorado, August 1979.

[McDaniel 77] McDaniel G. "METRIC: a Kernel Instrumentation System for Distributed Environments", Symposium on Operating Systems Principles, November 1977.

[Metcalfe 76] Metcalfe R. and David R., "Ethernet: Distributed Packet Switching for Local Computer Networks", Comm. ACM, July 1976.

[Mills 76] Mills D. "An Overview of the Distributed Computer Network", AFIPS Proc. NCC, AFIPS, June 1976.

[Millstein 1977] Millstein R. "A Distributed Processing System", Proceedings of the 1977 Annual Conference of the Association for Computing Machinery.

[NSW 76] NSW Protocol Commitee, "MSG: The Interprocess Communication Facility for the National Software Works", TR-3483, Bolt Beranek and Newman, December 1976.

[Nutt 72] Nutt G. "Evaluation Nets for Computer Systems Performance Analysis", Fall Joint Comp. Conf. 1972, AFIPS Proceedings, Vol 41.

[Parnas 72] Parnas D. "On the Criteria to be Used in Decomposing Systems into Models", Comm. ACM, Vol. 15, No. 12, December 1972.

[Peterson 77] Peterson G. and Fisher M. "Economical Solutions for the Critical Section Problem in a Distributed System", ACM Proc. of 9th Symposium on Theory of Computing", 1977.

[Peterson 79] Peterson G. "Understanding Concurrency", TR-59, Department of Computer Science, University of Rochester; also appeared as Ph.D Thesis, University of Washington, August 1979.

[Peterson 77] Peterson J., "Petri Nets", ACM Computing Surveys, Vol. 9, No. 3, September 1977.

[Piatkowski 75] Piatkowski Y. "Finite State Architecture", Techinical Report IBM, TR-29.0133, North Carolina, Research Triangle, August 1975.

[Postel 1974] Postel J., "A Graph Model Analysis of Computer Communications Protocols", Ph.D Thesis, University of California, Los Angeles, 1974.

[Riddle 78] Riddle W., Wileden J., Sayler J., Segal A., Stavely A. "Behavior Modeling during Software Design", IEEE Software Engineering, Vol. SE-4, No. 4, July 1978.

[Richards 68] Richards "BCPL Reference Manual".

[Rose 78] Rose C. "Measurement Procedure for Queueing Network Models of Computer Systems", ACM Computing Surveys, Vol. 10, No. 3, September 1978.

[Rovner 78] Rovner P. Private Communications on Reliable Transmission Protocol in RIG, Computer Science Department, University of Rochester, December 1978.

[Ryder 79] Ryder B., "Constructing the Call Graph of a

Program", IEEE Transaction of Software Engineering, Vol. SE-5, No. 3, May 1979.

[Schantz 79] Schantz R., "A Performance Analysis of National Software Works System", Bolt Beranek and Newman Inc., Report No. 3847, March 1979.

[Solomon 78] Solomon M. and Finkel R. "Roscoe-- A Multicomputer Operating System", TR-321, Computer Science Department, University of Wisconsion-Madison, September 1978.

[Sunshine 78] Sunshine C., "Survey of Protocol Definition and Verification Techniques", Proc. Computer Networks Protocols Symposium, Liege, Belgium, 1978.

[Sunshine 78] Sunshine C. and Dalal Y. "Connection Management in Transport Protocols", Computer Networks, Vol. 2, No. 6, December 1978.

[Tompson 1974] Tompson K. and Richie D., "The UNIX Time Sharing System", Comm. CACM, Vol. 17, No. 7, July 1974.

[Van-Mierop 79] Van-Meirop D., "Desing and Verification of Distributed Interacting Processes", PH.D Thesis, UCLA-ENG-7920, March 1979.

[Walden 1972] Walden D., "A System for Interprocess Communication in a Resouce Sharing Computer Network", Comm. ACM, Vol. 15, No. 4, April 1972.

[Watson 70] Watson R., "TimeSharing System Design Concepts", McGraw-Hill Book Company, 1970.

[Watson 79] Watson R. and Fletcher J. "A Protocol Structure for Network Operating System Services", Proceedings 4th Berkley Conference on Distributed Data Management, August 1979.

[West C. 1978] West C. "General Technique for Communications Protocol Validation", IBM Journal of Research and Development, Vol. 22, No. 4, July 1978.

[Witby-Strevens 78] Whitby-Strevens C. "Towards the Performance Evaluation of Distributed Computing Systems", Proc. CCMSAC 78, Chicago, IL, October 1978.

[White 76] White J. "A High-level Framework for Network-based resources sharing", AFIPS Proc. NCC, AFIPS, June 1976.

[Yoeli 78] Yoeli M. "A Structured Approach to parallel programming and Control", TR-126, Technion, Haifa, Israel, 1978.

Appendix A: Create a process

This appendix contains a description of the finite state machines and statistics in the form that is actually used by the performance monitoring system for the analysis of RIG. Section 4.1 describes the same example (the initialization of a virtual terminal in RIG) in a more descriptive form that is used throughout the dissertation.

Finite state machines are defined as BCPL procedures that are loaded together with the performance monitoring system. The differences between BCPL programs and the formalism used throughout the thesis is purely syntactical. A procedure call of a finite state machine definition produces a table of state transitions each containing either a message triple (which consists of a sender, receiver and message identifier) or a pointer to the lower level finite state machine. The statistics are calculated and presented for each state transition.

The following two pages contain programs ReadFile() and CreateProcess(). The statistics of those two models were sufficient to draw the conclusion that the file system accesses account for 25% of the total system in starting a terminal. The obvious optimization in this case is to keep the process definition table in memory as was described in Section 4.1.

The conclusion was drawn on the following basis: Execution of the transition FSM(ReadFile) is 2608 milliseconds. The system was idle 1312 milliseconds while waiting for the completion of the disk operation (there were no additional activities in the system). This constitutes about 50% of the overall execution time in creating a process (the accumulated execution time is 4878 and the total idle time is 1312 milliseconds). "OtherStatistics" accumulates statistics for all other events in the system that were selected by the CreateProcess descriptor. Again, about 50% of the total time was spent in creating new processes. Consequently, 25% of the total time was spent waiting for the file system to read process description tables.

```
and ReadFile() be
[
//  OpenFile:

Transition( "AnyProcess", "FileSystem", "OpenMsg")
BindMsgData("FileRequester", offset StatMessage.Sender/16)

//loop on  disk-I/O in order to open a file

let branch1 =
Transition( "System", "FileSystem", "DiskInterrupt")
Connect(branch1)

Transition( "FileSystem", "FileRequester", "FileOpened",
                                              branch1)

//loop on disk-I/O in order to read file blocks

let branch2 =
Transition( "FileRequester","FileSystem", "InputMsg")
Transition( "System", "FileSystem", "DiskInterrupt")
Transition( "FileSystem", "FileRequester", "OutputMsg")
Connect(branch2)

// close a file

Transition( "FileRequester", "FileSystem", "CloseMsg",
                                             branch2)
Transition( "System", "FileSystem", "DiskInterrupt")
Transition( "FileSystem", "FileRequester", "FileClosed")
Connect(LASTSTATE)

]
```

```
and CreateProcesss() be
[
// request to start a process

Transition("AnyProcess", "ProcessManager", "CreateProcessMsg")
BindMsgData("ProcessRequester", offset StatMessage.Sender/16)

// read PDT block:

FSMTransition("ReadFileEvents")

//create process map

Transition( "ProcessManager", "NewProc", "NewProcCreate")
Resume      ( "ProcessManager")

// Alternative I:  Requesting process receives a reply first

let branch1 =
Tansition( "ProcessManager", "ProcessRequester", "PMReply")
BindMsgData( "CreatedProcess", offset StatMessage.Data1/16)
Transition("AnyProcess", "CreatedProcess", "AnyID")
Connect(LASTSTATE)

// Alternative II: the newly created process is queued first

Transition("AnyProcess", "CreatedProcess", "AnyID",
                                                  branch1)
Transition( "ProcessManager", "ProcessRequester", "PMReply")
BindMsgData( "CreatedProcess", offset StatMessage.Data1/16)
Connect(LASTSTATE)

]
```

```
FSM: CreateProcess
   1) AnyProcess        => ProcessManager ,   CreateProcessMsg
      NumSamples:=  8      Execution:=    150   Overhead:= 50
      Delay:=   120

   2) FSM(ReadFileEvents)

      NumSamples:=   106  Execution:=   2608 Overhead:=  600
      Swapped:=    14      IdleWaitDevice:=   1312
      IdleWaitSwap:=  70  Delay:=  2030

   3) ProcessManager    => NewProc ,  NewProcCreateProcess
      NumSamples:=    8   Execution:=   1126   Overhead:= 50
      Delay:=   130

   4) ProcessManager    => ProcessManager ,   AnyID
      NumSamples:=    8   Execution:=    384   Overhead:= 50
      Delay:=   120

   5) ProcessManager    => ProcessRequester , PmReply
      NumSamples:=    8   Execution:=    128   Overhead:= 50
      Delay:=    60

   7) AnyProcess        => CreatedProcess ,   AnyID
      NumSamples:=    8   Execution:=    126   Overhead:= 50
      Swapped:=    13      Delay:=    110

   6) AnyProcess        => CreatedProcess ,   AnyID
      NextTransition:=     0
      NumSamples:=      5 Execution:=    160   Overhead:= 30
      Swapped:=    13      IdleWaitSwap:=  70   Delay:=    70

  10) ProcessManager    => ProcessRequester , PmReply
      NextTransition:=     0
      NumSamples:=      3 Execution:=     96   Overhead:= 20
      Swapped:=  22       IdleWaitSwap:=  100   Delay:=   260


AccumulatedStatistics:
NumSamples:=   146  Execution:=   4878 Overhead:=    900
Swapped:=    49     IdleWaitDevice:=   1312
IdleWaitSwap:=    240


OtherStatistics:
NumSamples:=   204  Execution:=   5610 Overhead:=    1330
Swapped:=   263     IdleWaitDevice:=   2750  IdleBUG:=    110
IdleWaitSwap:=  1450
```

Appendix B: BNF Definition of a Model

Syntax

```
<FSM-Model>       -> FSM-BEGIN <FSM-Name> <FSM-Def> FSM-END
<FSM-Name>        -> <Regular-FSM> ! <Indexed-FSM>
<Regular-FSM>     -> NAME
<Indexed-FSM>     -> NAME [ <INTEGER ]>

<FSM-Def>         -> <List-Tran>
<List-Tran>       -> <Tran> ! <List-Tran> <Tran>
<Tran>            -> <State-Label> : <Transition> ! <Transition>
<State-Label>     ->  NAME

<Transition>      -> <Event> <Next-State>
<Next-State>      -> <Implied-Next> ! <CONNECT> ( <State-Label> )
<Implied-Next>    -> < >
<Event>           -> <Message> ! <Tran-Index>
                     ! <Tran-Predicate> ! <Tran-FSM>

<Message>         -> <Sender> <Receiver> <MessageID>
<Sender>          -> NAME ! ANY
<Receiver>        -> NAME ! ANY
<MessageID>       -> NAME ! ANY

<Tran-Index>      -> INDEX ( <List-FSM-Name> )
<List-FSM-Name>   -> <FSM-Name> ! <List-FSM-Name> , <FSM-Name>

<Tran-Predicate> -> PREDICATE ( <List-FSM-Pred> )
<List-FSM-Pred>   -> <FSM-Pred> ! <List-FSM-Pred> , <FSM-Pred>
<FSM-Pred>        -> <FSM-Name> = <State-Label>

<Tran-FSM>        -> FSM ( <FSM-Name>  )
```

Variable Symbols:

<FSM-Model>, <FSM-Name>, <FSM-Def>, <Regular-FSM>, <Indexed-FSM>,
<List-Tran>, <Tran>, <State-Label>, <Transition>, <Next-State>,
<Implied-Next>, <Event>,
<Message>, <Sender>, <Receiver>, < MessageID>,
<Tran-Index>, <List-FSM-Name>,
<Tran-Predicate>, <List-FSM-Pred>, <FSM-Pred>
<Tran-FSM>

Terminal-Symbols:

:, [,], (,), ",", =,
FSM-BEGIN, FSM-END,
FSM, INDEX, PREDICATE,
CONNECT, ANY, NAME, INTEGER

Semantics:

Model

A finite state machine model is defined as a collection of transitions. A sequence of transitions implies a sequence of state-transitions; otherwise, the construct CONNECT explictly defines the next state. For each state alternate transitions are defined by preceeding each transition with the same state label.

FSM

The transition FSM occurs when the specified lower-level finite state machine passes from the initial state to the last state.

PREDICATE

The transition PREDICATE occurs when the specified list of finite state machines are all in a given state.

INDEX

The transition INDEX causes a change in system state. The index of all finite state machines specified in the list is decreased. A finite state machine modeling the message [i] in progress becomes a model for the message [i-1].

Appendix C: List of User Commands

The performance evaluator enters a command to the monitor which prompts for additional arguments. (The prompts are preceeded with the sign "=>"). Except for the command RUN, all other commands affect only the state of the monitor. The command RUN then interprets the state by actually performing the experiment.

The performance evaluator who uses finite state machines typically performs the following actions:

1) compose a file containing a finite state machine description;

2) load the file containing finite state machines (command LoadFSM);

3) receive raw statistics using either ReceiveTrace command which obtains statistics from a remote host or UserTraceDump which reads in statistics from a local file;

4) start the experiment with the command RUN.

Of course, steps 2-4 may be contained in a transcript file allowing the user to use just one command, UseTranscript. The following commands are supported by the monitor in RIG:

BINDPROCESS
=>Name:
=>ID:

This command binds a given process name to an integer. The monitor prompts for a list of pairs each consisting of a process name and an identifier. Symbolic names of processes enable the monitor to parse symbolic descriptions of finite state machines and to produce a trace of symbolic messages.

CLEANUP

In the case of a system crash, this command enacts a cleanup operation of the current experiment, saving all the statistics collected so far.

DISK
=>File:

This is a special purpose commmand for tracing disk activities. It produces a time-stamped trace of all disk commands and stores them in the specified file.

FSMLOAD
=>Trace (Yes or No)?
=>InputFile:
=>OutputFile:
=>OutputLevel:

This command loads descriptions of finite state machines contained in the specified file. The trace option, when enabled, produces a trace of all state-transitions. The output level parameter selects finite state machines having the specified or higher level within a hierarchy of finite state machines.

MESSAGETRACE
=>File:
=>Histogram:

This command produces a time-stamped trace of messages and stores them in the specified file. In addition, it produces a histogram of all the messages in the trace. (Which messages are traced is defined by another command: SelectTriples).

PROCESSTRACE
=>File:
=>Histogram

This command produces a trace of all processes with their run times. In addition, it produces a histogram of all the process run times. (Again, processes are selected according to the command SelectTriples).

READLOCATIONS
=>Address:

This is a special purpose command for reading arbitrary system locations. It is used to gather some gross system statistics such as the average number of queued messages in the system, or the average number of ready processes.

RECEIVETRACE:
=>Host:
=>Time:
=>File:

This command stores all statistics that are received from the remote host during the specified time period in a binary dump file (all other files are used in textual format). The command prompts for an identifier of the host being measured, the time of the measurement period (in seconds), and a name of the file that stores the raw data. The performance evaluator may choose to produce statistics directly without storing all statistics in the dump file.

RUN

This command runs the monitor. For example, if ReceiveTrace was initiated, the monitor collects raw statistics. If UseTrace was initiated the monitor gets statistics from the local file.

SELECTTRIPLES
=>Sender:
=>Receiver:
=>ID:

This command introduces the selection of messages and processes that appear in various traces. Only those processes and messages that match the specifications of stored triples are traced. The default value for each entry (a user hits "return" key) is the constant ANY, matching any value in the specified field of a message. Three "return" keys terminate the command.

SWAPTRACE
=>File:

This command produces a time-stamped trace of all swapped pages. It prompts for a name of the file that stores the data.

USETRACEDUMP
=>File:

This command uses the binary dump file that was created previously with the command ReceiveTrace. The binary dump file is used to produce textual files of traces of various events and statistics of finite state machines.

USETRANSCRIPT
=>File:

This command reads in the transcript file and interprets it as if the user were interacting with the monitor.

QUIT

Quit from the monitor.